Load Balancing using Docker Containers

Personnel
Ryan Caudill
Daniel McCade Freeman
Drew Jones
Colton Smith

CS499 University of Kentucky April 2017

Abstract

This project demonstrates the power and utility of Docker containers. These are software containers that use a provided image to deploy a virtualization of a complete file system. Docker containers are incredibly configurable and lightweight, allowing a new level of functionality and freedom beyond standard virtual machines. In this project, containers are used to deploy a large number of clients and servers with an intermediary load balancer. The load balancer acts as a reverse proxy in the network, handling and distributing client requests amongst the backend servers.

I. Introduction

The goal of the project is twofold. Firstly, design and implement a load balancer. Second, use Docker to spin up instances of servers and clients. These two design parameters are very apparent in the product. All aspects of the project are containerized using Docker and the load balancer is present. This was presented by the customer as a learning experience for the team; the motivation was to showcase this technology that is currently very popular in the software industry. As such, there is no real customer or user for the product besides those who are interested in seeing a basic demonstration of what Docker can do.

II. Product Requirements

The expectation of the project was to use Docker to create multiple instances of servers and clients and implement a load balancer to control traffic to the backend. The servers would be containerized and use web server technology – in this case, NGINX. Every server should supply the same content. The clients are to perform repetitive requests of different type (e.g., text and/or image pulls) to simulate server stress. The implementation must also log client and server statistics, such as the number of requests per client, number of requests to each server, etc.

III. Product Planning

III-A. Effort and Size Estimation

The estimated lines of code can be split among the three elements of the project: 50-100 lines of code for the clients, 30 for the servers, and 100 for the load balancer. In the end, our clients consisted of 54 lines of code, the servers used 68 lines, and the load balancer was 29 lines of code. The relative small size of the files is thanks to the Alpine Linux distributions; most of the code used is custom-built scripts to automate all functionalities of the project. Note: the images used to build the containers are not included in these sizing reports. The sizing reports are strictly based on custom scripts/files created and built on top of pre-made container files.

III-B. Schedule and Milestones

Date	Event
February 15th, 2017	Project Design started
February 19th, 2017	Client coding started
February 26th, 2017	Project Design completed
March 12th, 2017	Client code completed Client Testing started
March 15th, 2017	Client prototype available for customer review
March 26th, 2017	Server coding started
April 9th, 2017	Server code completed Server testing started Load Balancer coding started
April 16th, 2017	Load Balancer completed Load Balancer testing started Server/Client testing completed Project available for customer testing
April 21st, 2017	Load Balancer/Project testing completed

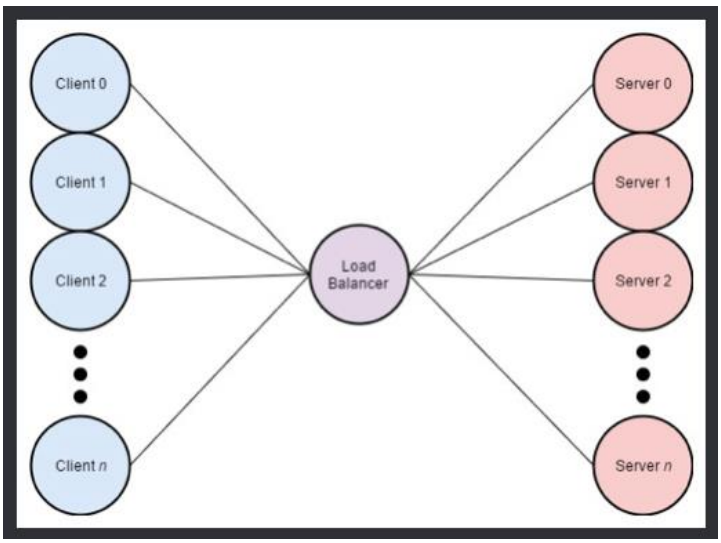
III-C. Platforms, Tools, and Languages

The client containers are the most restrictive: due to issues with certain commands, the clients require a native Linux/Mac environment. Specifically, the client module was run on macOS Sierra version 10.12.4, but it will work with earlier versions of Sierra as well. The load balancer and server containers allow more freedom. These are compatible and have been tested on Linux, Mac, and Windows 8.1/10. For Windows, Docker Toolbox must be installed along with VirtualBox to allow for port-forwarding for the load balancer and servers.

IV. Project Design

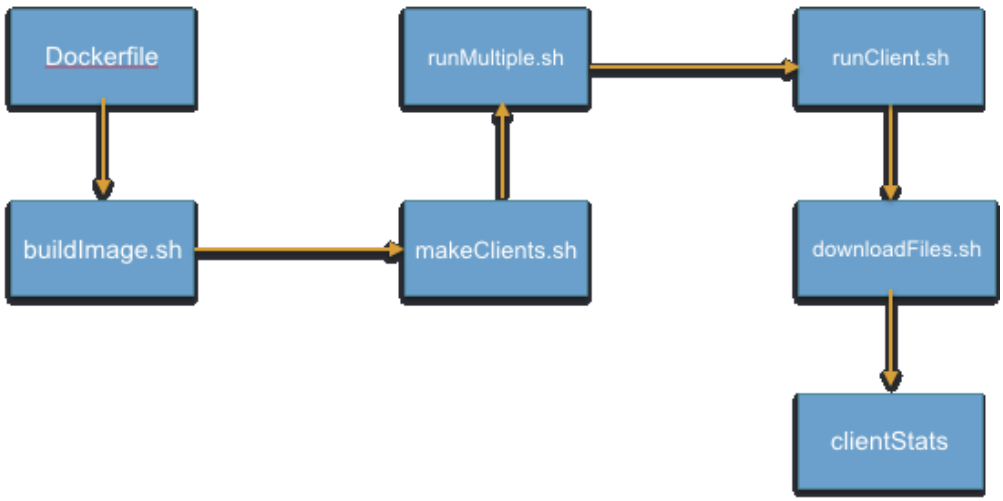
For the overall data flow, the project is divided into three modules: client, load balancer, and server. It is necessary to be able to spin up several clients and servers. The clients send their

requests to a single web server – in this case, the load balancer – which then forwards these requests to the backend servers, as per the load balancing algorithm.



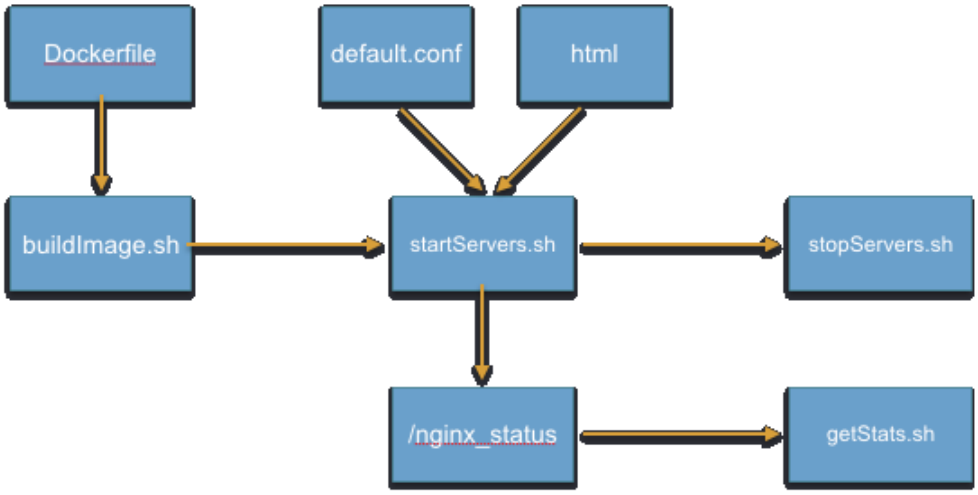
IV-A. Client

The client containers follow a rudimentary cycle of repeatedly sending requests to the servers, deleting the received file, and logging statistics.



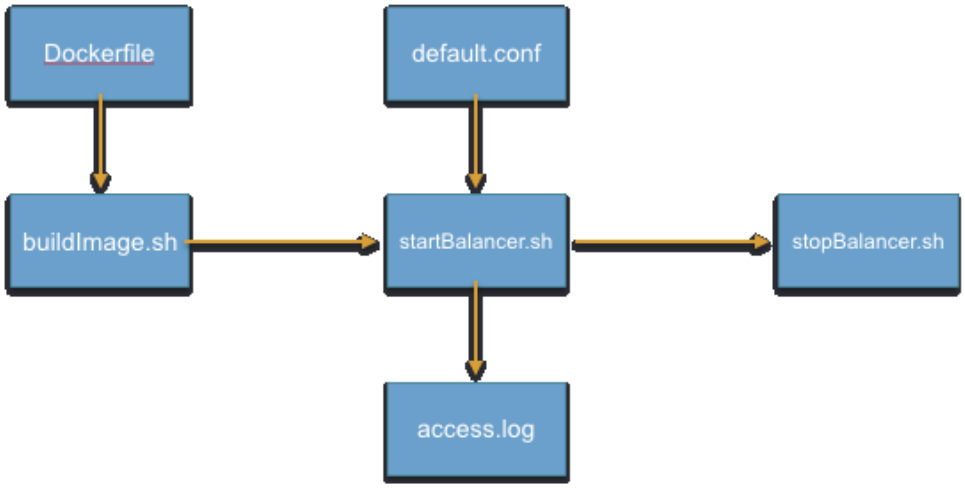
IV-B. Server

The servers listen for and promptly respond to incoming requests. Although these are client requests, they are coming directly from the load balancer. As the server containers send out content, they also log statistics for later analysis.



IV-C. Load Balancer

The load balancer is also containerized using Docker. Its purpose is to act as a reverse proxy and forward incoming client requests to the backend servers as per the given load balancing algorithm. In this project, two algorithms were implemented: round robin and least connections. Round robin forwards requests to the servers in strict sequence – first request to server 1, second request to server 2, and so on. Least connections will instead send each request to the server that up until that point in time, has received the least number of requests



V. Implementation

During implementation, a few problems caused setbacks in the project timeline and influenced the design implementations. The first issue encountered was running custom scripts inside of a

Docker container. More explicitly, passing a file to a script is not possible when giving a container a command in the Dockerfile. The simple solution to this is to write another script that will pass the file to the original script you want to execute, and tell the container to run the new script in the Dockerfile. In the case of this project, this causes no overhead or time delays.

Another problem encountered was the way in which Windows virtualizes linux based containers. Since Windows can not virtualize linux filesystems natively, Docker uses VirtualBox to create a private network for containers to run on while being executed on a Windows machine. This prevents any computer besides the host computer from accessing the servers/load balancer and any files they may be serving to meet requests from clients. The work around for this problem is to use port-forwarding to allow access to the host computers ports from outside of the virtual network created by VirtualBox. The set up for this is explained later in the User Manual section below.

One change was made to the design of the project due to time constraints. The original plan was to use Docker Swarms to implement custom load balancing algorithms, but this was not achieved due to the port forwarding issue encountered during server development. The change included using a configuration of the base server image used previously for servers (Nginx). By changing the default.conf file in /etc/nginx/conf.d/, it is possible to implement reverse proxying by giving a list of servers you wish to forward to. This also allows for out-of-box use of the round robin and least connections algorithms used in this project.

The only known restriction of the project is the use of custom client scripts on Windows 8.1/10. When running “runMultiple.sh”, an error is encountered which reads: “Proxy Error.” No solution was found for this.

V-A. Mac/Linux Test Results

Client	
Case	Pass / Handled / Fail
Multiple instances created	Pass
Instances run concurrently	Pass
Record data in log files	Pass
Send request to server	Pass
Close communications properly	Pass
Clean up files after closure	Pass
No server response	Handled

Server	
Case	Pass / Handled / Fail
Multiple instances created	Pass
Instances run concurrently	Pass
Record data in web page	Pass

Handle incoming requests	Pass
Respond with correct content	Pass
Request for unknown file	Handled

Load Balancer	
Case	Pass / Handled / Fail
Start up correctly on given port	Pass
Route incoming requests	Pass
Balance requests between servers	Pass

V-B. Windows Test Results

Client	
Case	Pass / Handled / Fail
Multiple instances created	Pass
Instances run concurrently	Pass
Record data in log files	Fail
Send request to server	Fail
Close communications properly	Fail
Clean up files after closure	Fail
No server response	Handled

Server	
Case	Pass / Handled / Fail
Multiple instances created	Pass
Instances run concurrently	Pass
Record data in web page	Pass
Handle incoming requests	Pass
Respond with correct content	Pass
Request for unknown file	Handled

Load Balancer	
Case	Pass / Handled / Fail
Start up correctly on given port	Pass
Route incoming requests	Pass
Balance requests between servers	Pass

VI. Future Enhancements

There are a few future enhancements for the project that would be beneficial in scaling or implementing the product for production. The first would be creating client script compatibility for Windows computers. Of course, this is not 100 percent needed as the servers could run on a macOS or linux computer; this would however allow for a more widespread customer base. The next enhancement would be the implementation of custom load balancing algorithms. This will allow for a more specialized form of load balancing to fit the needs of the customer. For this projects purposes, round robin and least connections were more than enough, but a custom algorithm may be needed for scalability and reliability. Finally, the use of more lightweight server and load balancer images would benefit the speed of each, and would also allow for more instances of both to be spun up while reserving the host computers resources. Docker is about being small, quick, and efficient, but the current size of the server/load balancer containers is roughly 180MB. There is an Alpine distribution of Nginx which is ten times smaller, but due to time constraints it was not implemented into our project.

VII. Conclusion

The project was presented as more of a learning opportunity than a full-fledged product. In this way, it was a huge success for the team. As a relatively new technology, Docker is gaining traction and booming in popularity in the software industry. As such, learning to use it in such a way along with the bonus of load balancing and web server technology has been a very enriching experience. The customer was also impressed with the implementation which was presented to him later in the semester. It met his expectations in most aspects and in some ways, the project exceeded expectations.

VIII. References

https://www.docker.com/
https://training.docker.com/category/self-paced-online
https://www.nginx.com/resources/admin-guide/load-balancer/

IX. User's Manual and Installation Guide

IX-A. Installation Guide

For the entirety of this project, Docker version 17.03.X must be installed and running at the time of use. While running on Windows 8.1/10, follow the official installation guide for the Docker Toolbox and choose the installation option to use VirtualBox to allow for simple port-forwarding: https://www.docker.com/products/docker-toolbox. For Linux/macOS, follow the official Docker installation: https://store.docker.com/editions/community/docker-ce-desktop-mac?tab=description. To create the custom images created for the project, follow the guides listed below for the clients, servers, and load balancer.

IX-B. User's Manual

Clients:

Files included:

cleanClients.sh: This will navigate into all subdirectories and clean out the log files generated by running the clients.

client-alpine (directory):

- clientStats (directory): contains log files generated by the client
- Dockerfile: contains configurations for client container
- downloadFiles.sh: downloads files in a for loop, deletes them, and appends download speed to log file inside of clientStats with a filename generated by the time and date the client was run.
- runClient.sh: runs the client by passing url to downlaodFiles.sh
- url: a text file containing the url that the client will download from

makeClients.sh: Requires input; will create "n" copies of client-alpine based on user input. These copies are roughly 18 KB so this creates no problems for the host OS.

runMultiple.sh: Requires input; will run "n" copies of client-alpine as long as "n" copies exist. This will run the client container and mount the containers directory to the local directory created by makeClients.sh. This allows us to retrieve the log files after the container finishes and shuts down.

client-alpine/buildImage.sh: This will build the client image using Docker.

Dockerfile: Contains the information needed for Docker to build a custom client image.

Instructions for use:

Docker version 17.03.X must be installed and running at the time of use. First, you must build the client image using the provided Dockerfile. To do so, run buildImage.sh. Then, run makeClients.sh to create the number of clients you wish to use (our tests have shown that 100 clients is the maximum that could run on a 2011 MacBook Pro). Next, run runMultiple.sh to instantiate containers and retrieve results. For both makeClients.sh and runMultiple.sh, prompts are given to the user. Log files can be found inside of each containers directory.

Servers:

Files included:

html (directory): This file contains the html files that will be used by all servers. All servers that are spun up will reference this one directory to ensure consistent responses to the client.

Dockerfile: Contains the information needed for Docker to build a custom client image.

startServers.sh: This bash script will allow the user to enter the number of server instances they wish to spin up. It will ask for the number of servers the user wishes to instantiate as well as which port number it would like to begin at. Then, it will creating the servers such that their address corresponds to the starting port as well as the ports immediately after.

stopServers.sh: This bash script will allow the user to enter the number of server instances they wish to shutdown. It will ask for the number of servers the user wishes to shutdown, as well as the starting server number.

getStats.sh: This bash script will obtain the statistics of the individual Nginx servers as well as the load balancer if need be. It will ask for the number of servers you wish to obtain the stats of, the IP address of the servers, and the port number in which the series of servers begins.

serverStats (directory): This folder contains the statistics of each server after it has been run.

default.conf: This configuration file allows you to customize the Nginx server. All that we have changed is the ability to access statistics. This allows you to go to http://IP.Address:port/nginx_status to view live statistics while the server is running.

Instructions for use:

Docker version 17.03.X must be installed and running at the time of use. First, you must build the server image using the provided Dockerfile. To do so, run buildImage.sh. Then, run startServers.sh and input the number of servers you wish to spin up. It will also ask for the starting port you wish to use. It will then create servers with names "server1" - "serverN"(where N is the number of servers you started). When you are finished running the servers and want to shut them down, run stopServers.sh and enter the number of servers you wish to stop and the starting server number. To obtain stats after you have finished running clients, simply run getStats.sh and follow the prompts for the number of servers you want the stats of, the IP address of the servers, and the port number in which the series of servers begins.

Load Balancer:

Files included:

buildImage.sh: This will build the docker image if one does not already exist. MUST BE RUN FIRST.

Dockerfile: Contains the information needed for Docker to build a custom client image; used by buildImage.sh

startBalancer.sh: This bash script will spin up the load balancer and direct client requests to server.

stopBalancer.sh: This bash script will allow the user to stop the load balancer. This only needs to be done if you wish to configure the load balancer image or stop it permanently.

Instructions for use:

Docker version 17.03.X must be installed and running at the time of use. First, you must build the server image using the provided Dockerfile. To do so, run buildImage.sh. Then, run startBalancer.sh and the load balancer will spin up. It will then create a docker container named “load-balancer” that will be running on port 8080 of the localhost. This can of course be changed in the startBalancer.sh script. When you are finished running the balancer and want to shut it down, run stopBalancer.sh.s