UTILIZING THE PORTABILITY OF DOCKER TO DESIGN A PORTABLE AND SCALABLE CONTINUOUS INTEGRATION STACK

Jordan Goetze
Computer Science
North Dakota State University
Fargo, North Dakota 58103
jordan.goetze@ndsu.edu

Abstract

With the release of Docker as an open source project in March of 2013, the project has gained substantial pull in the world of systems administration, and especially in software development. The removal of the hypervisor layer, as well as the reduction of unnecessary dependencies in Docker based virtual machines - commonly called images - allows for extremely reduced storage footprints and effectively (as explained on the project's website) allows "'dockerized' projects to be completely portable"("What is Docker?"). This portability makes for a powerful tool when designing continuous integration stacks.

When designing a continuous integration system, it is important to consider several important factors, such as the kinds of scenarios the system needs to handle, what sorts of assumptions about the environments need to be made, and how configurable a system needs to be in order to handle unforeseen edge cases on the fly. A Docker based system could, in theory, help in solving these issues if used properly. Instead of moving source code between test and production environments, you could instead transport the entire environment to the production location through the use of Docker images. In addition, because the “production code” is tied to a “production environment” which is encapsulated in a Docker image, it becomes unnecessary to make assumptions about the states of production deployment environments, and making tweaks to the environment is a simple

matter of editing a Dockerfile[1] and rebuilding the image.

Due to the fact that Docker images are sandboxed virtual machines, and because Docker has a single inheritance based hierarchy[2] it is possible to run a Docker image inside another Docker image. When we examine the rather traditional model for continuous integration servers, wherein there is a service that acts as a master or dispatcher - providing tasks to an n-series group of slave/ worker instances tasked with running build tasks and tests, we can see that both the master and the slaves could potentially benefit from the portability and also the transience of docker images. Taking queue from Martin Fowler's article on designing phoenix like servers[3] to avoid what he calls configuration drift[4]. By implementing this mannerism, we can create continuous integration environments with Docker that are portable and scalable. The portability coming from the nature of Docker images, and the scalability by spawning up several instanstances of a docker image[5] containing an implementation of a worker. Thusly, we have a pattern for transporting and scaling the relatively complex systems and processes that make up a build server. From here the next step would be to design a pattern for recursively inheriting Docker containers as well as defining states that individual images should provide such that this pattern would function as intended.

---

1 - A configuration file which Docker uses to generage images

2 - A new docker image must be built from another existing image, or from a base image, base images are images that provide the bare minimum for an environment. For example ubuntu:14.04 is a base image that contains a functional ubuntu environment.containing an implementation of a worker. Thusly, we have a pattern for transporting and scaling the relatively complex systems and processes that make up a build server. From here the next step would be to design a pattern for recursively inheriting Docker containers as well as defining states that individual images should provide such that this pattern would fun12ction as intended.

3 - “... it is a good idea to virtually burn down your servers at regular intervals. A server should be like a pheonix, regularily rising from the ashes”(Fowler).

4 - “ad hoc changes to a system[']s configuration that go unrecoarded”(Fowler).

5 - A single running inst
ance of a Docker image is called a container as it represents a sandboxed runtime environment based on the original Docker image. Think of an image like a virtual machine image, a container like a clone of that snapshot that runs independantly of the original image and results in no change to the original image when mutations are performed on the container.

Docker and Friends

The very recent release of the docker-machine, and docker-swarm tools have made creating and using Docker environments even easier, and even more cross-platform than even a few weeks ago. Do note however that at this point in time, many of the following features and tools are still expirimental. The first and most notable is the docker-machine command line interface. This tool allows for the creation of local and remote instances of docker containers, and also allows for Windows support for Docker. The way this tool works is be creating virtual machines and installing the Docker environment inside of it. The tool supports a few different drivers currently, one of which being Virtual Box. Using docker-machine with the Virtual Box driver[6] results in the creation of a Virtual Box virtual machine with the Docker system installed and ready for work. The encapsulation of the Docker system in an already cross platform virtual machine system, means that Docker can now be natively used on Windows[7] docker-machine also provides container creation support for docker-swarm. docker-swarm is another tool that allows for efficient, local and remote scaling of containers. The current workflow for using docker-swarm is to designate a “master” host called a “swarm host” by docker-swarm. From there “slaves” termed swarm nodes (or just nodes). Once the swarm host and and nodes are set up, issueing commands to the swarm host, will result in dispersal of the command to all nodes that are a part of the swarm. So if there are 15 nodes in the swarm, and a command to run a web server is sent to the swarm host, in a matter of minutes (sometimes seconds depending on container availablity on each node) there will be 15 instances of the web server running the same server. This works very well for systems which have a central server that each of the nodes can look to for work, because as of today, individual nodes are incapable of comminucating with each other. The developers of docker-swarm have noted that they have plans to allow communication between nodes.

Boundary Free Developement

Having hopefully etabilished several of the possible utilities that Docker provides, we can now discuss a slightly more specific

---

6 - This requires virtual box to be installed on the operating system of your choice
7 - Additionally, Microsoft has promised additional support for Docker, on future Windows Server iterations, possibly even to the point on hosting Windows in a docker container

application for Docker and its utilities: a continuous integration(CI) stack. Traditional methods of creating a CI stack usually involve a lengthy process of installing the “master” server, and optionally “slave” servers that serve as workers for building and shipping software releases. Installing the master is often a difficult task, requiring much configuration. Installing slaves is equally as much work, depending on the hosting environment. Assuming that the slaves are not hosted in virtual machines, each slave would need to be manually installed and configured identically. Assuming the slaves are hosted in virtual machines, ip addresses, port routing and the such would still take a decent amount of effort. In summation, traditional CI systems are a fairly complex and fragile topic, and the complexity here does not even begin to account for hidden complexity introduced by different systems, as well as the manging and backing up those servers (speciffically) the master server. Traditional CI servers are not simple task to set up.

Simple, Reproducable, Master

The “master” server setup is the simplest case here, all that is needed here is to create a Dockerfile for the image, and then build the image and run it in a container. A Dockerfile is a configuration file that defines several important things about a Docker image. Because Docker relies on a single inheritance herarchy, the dockerfile must define a base image from which to build the newly defined docker image. A good example of this is ubuntu:12.04, a very common definition for a base image. It is also required that we delegate a maintainer for the image. From here on out we can define operations that should be run on the image when it is being built. One of the most common of these commands is the “RUN” command. The RUN command allows you to run commands inside the container as it is built. Examples of this would include:

```
RUN apt-get update && apt-get upgrade && apt-get install
curl
```

This command will run the chain of apt-get calls inside the container, resulting in the installation of the curl executable inside the container. Docker builds images using intermediate steps, so after each line defined in the dockerfile, the state of the container will be saved as an intermediate image[8]. As another example to create a container that

8 - when the build is completed successfully intermediate images are deleted

sets up a basic node http server the dockerfile might look like:

```
FROM node
MAINTAINER <your name> <your.email@provider.ext)

RUN npm install http-server
ADD ./src /data/src
CMD http-server /data/src
```

In this example we define node as the base image, set a maintainer, and then use npm[9] to install http-server[10]. We also use a new few commands: “ADD” and “CMD”. ADD tells docker to copy the directory structure pointed at by the first operand which is relative to the dockerfile on the host, and to copy it to the path inside the container defined by the second operand. CMD sets the default command to be run inside the container when docker starts it up. In this case we tell http-server to run in the /data/src directory. Assuming our master is no simpler than a javascript client making calls to it's workers, all we need to do now is build it.

```
docker build -t my-master .
```

Should build the image and

```
docker run my-master
```

should start the container in detached head mode. The “docker build” command builds the image, the “-t my-master” means to give it a reference tag called “my-master”, and the “.” means that we should search the current directory for the dockerfile to use in building the image. For the second command, the “docker run my-master” tells docker to run the “my-master” image as a container. From there the my-master will run silently in the background. We can take this dockerfile and the code used to build it and move them to an entirely different host, and the environment that the http-server runs in will be identical to the one we just made.

Distributed, Swarmed, Workers

---

automatically
9 - node package manager
10 - a basic http-server implemented in nodejs

Docker introduces an easier way to host a CI server with distributed workers. docker-machine takes away much of the pain of configuring and creating hosts. With a series of commands a simple as:

```
docker run swarm create
0a7ad347861277c3ea4eacc44e9d34e2

docker-machine create -d virtualbox --swarm --swarm-master
--swarm-discovery=token://0a7ad347861277c3ea4eacc44e9d34e2
swarm-master

docker-machine create -d virtualbox --swarm --swarm-
discovery=token://0a7ad347861277c3ea4eacc44e9d34e2 swarm-01

docker-machine create -d virtualbox --swarm --swarm-
discovery=token://0a7ad347861277c3ea4eacc44e9d34e2 swarm-02

docker-machine create -d virtualbox --swarm --swarm-
discovery=token://0a7ad347861277c3ea4eacc44e9d34e2 swarm-03
```

These commands will set up a swarm host with 3 nodes connected to it[11]. In the section:

```
--swarm-discovery=token://0a7ad347861277c3ea4eacc44e9d34e2
```

The hash following the token:// is a part of the magic of docker-swarm. This token is unique to each docker swarm, and is how docker-swarm connects and routes information between the swarm master and the nodes. The first line:

```
docker run swarm create
```

returns the second line:

```
0a7ad347861277c3ea4eacc44e9d34e2
```

The “docker run” part is part of the docker client command line tool. And means that the next token (this being “swarm”) will be a base

11 - in this example the host and nodes will run in local virtualbox virtual machines

image for a docker container. If the image has not already been downloaded previously, docker will download it and the proceed to the next part. The “create” keyword is an argument to be passed to the default command run in the docker container based off the “swarm” image. In this case the default command is the swarm client running inside the swarm container. Reguardless of how things get passed into the docker container for the swarm image, the swarm client running inside it will make a connection to the Docker Hub[12], and sets up a new swarm registry. This command returns a token corresponding to the swarm created in the Docker Hub. This token is used by the nodes to tell the Docker Hub their ip's and ports, and for the swarm master to ask the Docker Hub for the ip's and ports to the nodes. The lines following the “swarm create” utilize docker-machine to create four virtualbox virtual machines. Three of the four are nodes and the last is the swarm master. No more configuration should be nessisary to set up communication between the swarm and the nodes, anything that needs to be done on the nodes can now be done using the docker remote api Which is something that looks like:

```
docker -H <swarm-master-ip>:<port> info
```

Because all of the nodes are hidden behind the swarm-master, everything will seem as though you were using a locallhy hosted docker instance, with only one image running solely due to docker-swarm. From here all it would take to install the worker software to the swarm would be a simple:

```
docker -H <swarm-master-ip>:<port> run my-worker
```

And suddenly we have a small army of distributed workers running the my-worker image. An ideal function for the containers running the my-worker image would be to ask our CI master server for work to do.

Conclusion

Docker allows a CI stack that could traditionally take days to set up properly, to take a matter of hours for beginners to figure out. And the real clinch pin here is the --priveleged option that docker provides. This comand line switch to the docker client, allows for docker

12 - the docker hub is a repository for docker base images and also a regiser for tracking swarms

instances to run inside of other docker instances. This means that for testing, the worker nodes that we created earlier, can test code in docker containers, to replicate complex environments that are identical to the production environements because docker images encompass not only the source code, but also the ideal runtime environment for that code to be deployed under. Additionally, tools like docker-compose, another docker tool whose usage will be left an exercise for the reader, can alow for docker to scale hosted instances of a swarmed worker client with:

```
docker-compose scale 500
```

Which would result in the creation of new worker processes distrubuted as needed across nodes until the total pool equals 500. With one simple commnd, you can start and stop all the nodes of a swarm, with another couple commands, you can start, stop, and redeploy code to all nodes. The possibilities are only limited by the number of hosts you can access, and the system resources of those hosts. The promising future of docker and similarlily promising new tools mean an exciting future for Docker users. But it also means that we can look forward to more efficient implementations of already great patterns that would benefit a CI server, making config hassle a one time ordeal, allowing environments to be moved and migrated painlessly, and effortlessly.

References

“Announcing Docker Machine Beta”. n.p. 26 Feb. 2015. Web. 30 Feb. 2015.

“Dockerfile Referene”. n.p. n.d. Web 6 Feb. 2015.

“Docker Machine” n.p. n.d. Web. 31 Feb. 2015.

“Docker Swarm” n.p. n.d. Web. 31 Feb. 2015.

Fowler, Martin. “PheonixServer”. n.p. 10 July. 2012. Web. 6 Feb. 2015.

“Orchestrating Docker with Machine, Swarm, and Compose”. 26 Feb. 2015. Web. 30 Feb. 2015.

“Scaling Docker with Swarm”. n.p. 26 Feb. 2015. Web. 30 Feb. 2015.

"What is Docker?". Docker. Docker Inc. n.d. Web. 6 Feb. 2015.