An Assessment of the Docker Micro Container

Darrell Pratt, Architect Leader, GBS

Executive Summary

Virtual machines have always been a large part of any sizeable development organization and their use as a means to virtualize the resources of a physical machine are well known. A new application called Docker is turning this concept of virtualization into immediately accessible micro containers for any organization to easily tie into their workflows. This framework has grown verey rapidly with even Amazon Web Services just recently announcing support for Docker in their Elastic Cloud service.

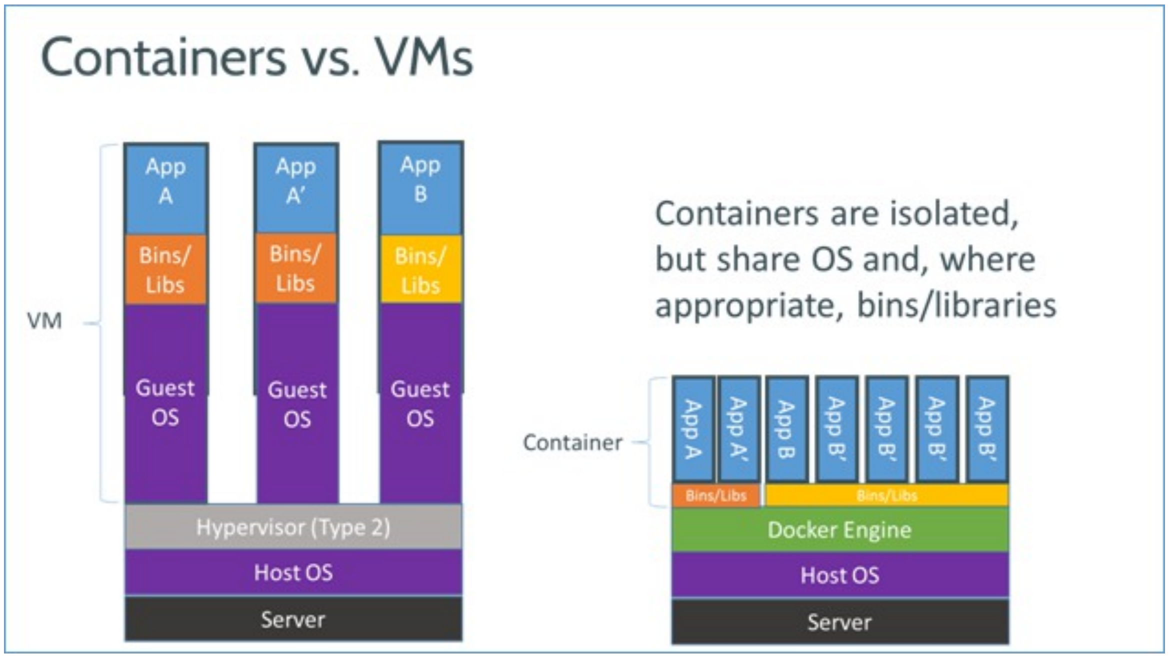
Problem Statement

Virtualization has helped companies make the best use of their physical hardware within data centers. Primarily, this has been the realm of VMWare and few other built in solutions within various operating systems. Although some groups have built quite a bit of tooling around VMWare and the provisioning of images to users, there is still quite a bit of setup involved with this approach and no true method for the developer to declare which resources are needed for the image and how to separate those images based upon the application's needs. At the completion of provisioning, one still has a VM that looks like a physical machine that might be running various application servers or databases. There is no true separation of concerns with this approach.

Docker Background

Docker was created to solve the issue of build once, run anywhere. Docker automates the creation of a self-sufficient container that can run anywhere, but displays the same outward behaviour to the developer or ops team regardless of location. A central difference between Docker and traditional virtual machines, is that the Docker image is extremely lightweight as it has no host OS (it uses the host OS) and it can be deployed and started with very minimal delay. Docker itself uses LXC (Linux Containers) which acts as a direct interface to the containment features which are part of the linux kernel. This acts at a lower level than Xen which (a hypervisor) and much lower than the VM level such as VMWare uses. Docker also uses *cgroups* which stands for control groups. Using cgroups allows the container to control the amount of physical resources that the container can use (RAM, System Cache, CPU prioritization). In order to virtualize the storage subsystem, docker uses UnionFS. This file system at the simplest definition

An illustration of the difference between a traditional container and Docker is illustrated below.



Docker vs. VMs

Docker manages to scale in the sense of maintaining many images of different applications or version of those applications because of this design pattern. The container is essentialy the application and its dependent binaries and libraries. Docker is intelligent enough to only store the differential between any of these layers across the same base images.

What does this mean? For example, imagine a large Java Application which is maintained in git with a build process hosted within Jenkins. The build process could easily take a given base Dockerfile and load a different codebase into that container, instantiate it, run a full test suite and then decommission that container. The build could do this is a completely automated fashion. These containers can also run in parallel on a host or host sharing the unchanged components between each container. This allows for scaling the number of containers with very little disk space or resources being used. With a traditional VM infrastructure, this would require unique guest host images, which are several gigabytes in size each and take time to build and startup. Docker makes this trivial and extremely fast.

The Docker name itself comes from the idea of shipping. As the company explains it, shipping used to be done in a break bulk fashion. Simply put, cargo was shipped as it came, without any standardized boxing. Shipping companies needed to worry about interactions between any items that might be onboard a ship at any time. This was revolutionized after 1960 as shipping containers were introduced. These containers are common sight now, but this one change allowed for effeciencies in loading and unloading of cargo, and relay of those items through the other transit systems. This analogy holds true to what Docker is attempting with system virtualization as well. Docker generally attempts to alleviate the concerns over what might be contained within. This containment solves many problems with library and dependency hell that are typically seen with standard virtualization and certainly bare metal deployments. This allows any application and its various dependencies to be packaged up as a lightweight container that can be deployed virtually anywhere.

That should suffice for a background of Docker and its goals. The power of Docker is truly its ease of use and configurability. The following sections will explore how to setup a Docker image and run a container as well as some use cases for what Docker could be used for at Nielsen.

Configuration

Docker makes use of a simple configuration file to build images. The file is plain text and has a fairly small set of commands. The configuration is a set of instructions. These instructions are all in the form below.

```
# comments being with hash
INSTRUCTION arguments
```

The first instruction in any dockerfile must be the FROM instruction. This tells docker which image is considered to be the base image of the file. Think of this as a state at which you wish to further customize the container.

```
# Start with a base Ubuntu image
FROM ubuntu:13.03
```

This would use the 13.03 release of Ubuntu as our base image. There are several distributions one can choose from, such as centos, smartos and many others.

After specifiying the base image with the **FROM** instruction the container more than likely needs to be customized with your preferred libraries and binaries needed to run your application. The **RUN** instruction is used here to run any arbitrary command on the new image.

```
# Start with a base Ubuntu image
FROM    ubuntu:13.03
# Load some libraries
RUN     apt-get install -y python-software-properties python g++
RUN     apt-get install -y make software-properties-common
```

The above dockerfile will load Ubuntu and then run apt-get to install a few libraries onto the new image. Note the ***-y*** on **apt-get**. This is important so as to make the apt-get command run non-interactively.

The RUN instruction can also take the following form.

```
RUN ['command', 'param1', 'param2' ...]
```

This is useful when running a command on the image that takes a complex set of arguments. In our case above it is not necessary and the first form appears a bit cleaner to the author. To add to our example above, we will add additional RUN instructions to build out a nodejs environment for our new image.

```
# Start with a base Ubuntu image
FROM    ubuntu:13.03
# Load some libraries
RUN     apt-get install -y python-software-properties python g++
RUN     apt-get install -y make software-properties-common
#Add node repository to sources.list and update apt
RUN     add-apt-repository ppa:chris-lea/node.js && apt-get update

#Install node.js
RUN     apt-get -y install nodejs
```

With 10 lines of text, we now have an ubuntu machine that will have nodejs installed and ready for use. We still need to get our application onto the image however. The **ADD** instruction allows us to mount resources onto the image. This command takes the following form.

```
ADD <src> <dest>
```

Any number of these ADD instructions can be in the dockerfile. Note that the files are added to the new image as 755 0:0. This is important to understand if your application has any complex file system permission requirements. With the ADD instruction we can now add in our application from the local machine into the new image. Assume that our dockerfile is at the project root of our application then the following will add our application's files to the new image.

```
# Add project from current directory to /src dir on image
ADD . /src
```

As we are building a nodejs image we can now use the RUN command to add our node modules needed for the application using npm. (If you have not used node or npm, go do it now). The following instructions will install our dependencies on the image.

```
# Install app dependencies
RUN cd /src/server; npm install
# Install nodemon to run our application forever
RUN npm install nodemon
```

Now that we have our image set to run our application, the next important topic is that of port mapping and linking. Linking is a bit complex so it will be covered later in the document, but port mapping is straight forward. Our application we want to run in our container will listen on some port (if it is a network application). In order to map that internal port of the container to the host OS, we use the **EXPOSE** command as follows.

```
# Expose our container port 300 to the host
EXPOSE 3000
```

Note that we have only specified that this port is available to the host OS. In order to map from our host port to this exposed port that will be specified when we start the container. That will be covered in the command line section later in this document.

The final instruction to use in this example is **CMD**. This instruction takes the following form.

```
CMD ["executable", "arg1", "arg2", ...]
```

This instruction can only exist once within the dockerfile as this is informing the container to run this command and then wait for connections to the container. We've basically created an image that will load the OS, libraries needed, applications needed, loaded our own application and then started our application and exposed it to the host OS in a small amount of configuration code. The full example is below.

```
# Start with a base Ubuntu image
FROM    ubuntu:13.03
# Load some libraries
RUN     apt-get install -y python-software-properties python g++
RUN     apt-get install -y make software-properties-common
#Add node repository to sources.list and update apt
RUN     add-apt-repository ppa:chris-lea/node.js && apt-get update

# Install node.js
RUN     apt-get -y install nodejs

# Map our local files to the image
ADD . /src

# Install app dependencies
RUN cd /src/server; npm install
# Install nodemon to run our application forever
RUN npm install nodemon
# Expose our nodejs port to the host
EXPOSE 3000
# Run our server using nodemon
CMD ["nodemon", "/src/server/server.js"]
```

There are few other instructions that were not used in this example. The **ENV** instruction allows the container to be set with any set of environment variables. ENV takes the following form.

```
ENV <key> <value>
```

Note that while this could be helpful for a set of well known environments settings, one can pass environment key value pairs to the container on running as well.

The **ENTRYPOINT** instruction is similar to the **CMD** instruction, but it is used to basically tell docker to run the image as an executable. There can be more than one **ENTRYPOINT** command in a dockerfile but the last one is the only one which will run. This instruction takes the following form.

```
ENTRYPOINT      <cmd> <args>
```

A possible example would be to use an **ENTRYPOINT** instruction to create a container that would run a data import process. This would allow any number of these containers to be activated when needed and to be torn down and cleansed from the system when not in use.

Now that we have written a dockerfile to configure an image, we need to actually build that image to use it. The next section will detail the docker commands and give some guidance on how to setup an environment to work with docker effectively.

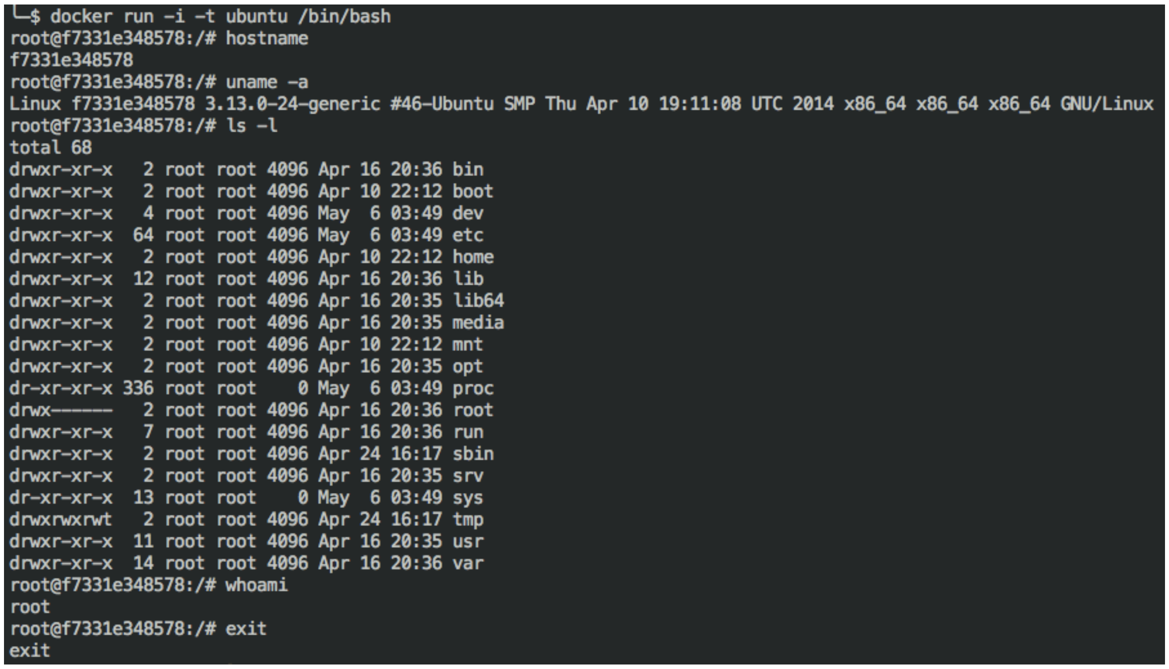
Image Management and Creation

The docker client and daemon process are easy to install on a number of operating systems. The following set of commands will install docker on an unbuntu or ubuntu like system.

```
$ sudo apt-get update
$ sudo apt-get install docker.io
$ sudo ln -sf /usr/bin/docker.io /usr/local/bin/docker
```

If you have made it this far, note that most examples will start to look repetitive with the use of *sudo*. Docker does need to be run as root or in a privelaged group, but if a docker group exists on the host OS (which it should if it is installed properly) then if the user (you in this case) is a member of the docker group then sudo is not required. This is highly recommeded for learning docker, but in production one would probably want to run with sudo to preserve security.

Once docker is installed you can test the installation by running the simplest container possible. The following command will run a base ubuntu container and give you an interactive shell to run commands from. Note the load time of the container upon the first run and then exit the container and run it again and note the difference. This is the effect of image caching and this will begin to demonstrate the speed of docker.



```
└─$ docker run -i -t ubuntu /bin/bash
root@f7331e348578:/# hostname
f7331e348578
root@f7331e348578:/# uname -a
Linux f7331e348578 3.13.0-24-generic #46-Ubuntu SMP Thu Apr 10 19:11:08 UTC 2014 x86_64 x86_64 x86_64 GNU/Linux
root@f7331e348578:/# ls -l
total 68
drwxr-xr-x   2 root root 4096 Apr 16 20:36 bin
drwxr-xr-x   2 root root 4096 Apr 10 22:12 boot
drwxr-xr-x   4 root root 4096 May  6 03:49 dev
drwxr-xr-x  64 root root 4096 May  6 03:49 etc
drwxr-xr-x   2 root root 4096 Apr 10 22:12 home
drwxr-xr-x  12 root root 4096 Apr 16 20:36 lib
drwxr-xr-x   2 root root 4096 Apr 16 20:35 lib64
drwxr-xr-x   2 root root 4096 Apr 16 20:35 media
drwxr-xr-x   2 root root 4096 Apr 10 22:12 mnt
drwxr-xr-x   2 root root 4096 Apr 16 20:35 opt
dr-xr-xr-x 336 root root    0 May  6 03:49 proc
drwx------   2 root root 4096 Apr 16 20:36 root
drwxr-xr-x   7 root root 4096 Apr 16 20:36 run
drwxr-xr-x   2 root root 4096 Apr 24 16:17 sbin
drwxr-xr-x   2 root root 4096 Apr 16 20:35 srv
dr-xr-xr-x  13 root root    0 May  6 03:49 sys
drwxrwxrwt   2 root root 4096 Apr 24 16:17 tmp
drwxr-xr-x  11 root root 4096 Apr 16 20:35 usr
drwxr-xr-x  14 root root 4096 Apr 16 20:36 var
root@f7331e348578:/# whoami
root
root@f7331e348578:/# exit
exit
```

docker run commander

In the above example we use the docker client to run a base image in interactive mode using the bash shell. Once this container has run and used then it is cached for faster startup on subsequent invocations.

The Docker Client

The docker client is run through the docker cli. This command acts as a client to the dockerd process which has the responsiblity of controlling the actual container. Below are the list of command options for the docker cli.

```
docker
Usage: docker [OPTIONS] COMMAND [arg...]
 -H=[unix:///var/run/docker.sock]: tcp://host:port to bind/connect to or unix://path/to/sock

A self-sufficient runtime for linux containers.

Commands:
    attach    Attach to a running container
    build     Build a container from a Dockerfile
    commit    Create a new image from a containers changes
    cp        Copy files/folders from the containers filesystem to the host path
    diff      Inspect changes on a container's filesystem
    events    Get real time events from the server
    export    Stream the contents of a container as a tar archive
    history   Show the history of an image
    images    List images
    import    Create a new filesystem image from the contents of a tarball
    info      Display system-wide information
    inspect   Return low-level information on a container
    kill      Kill a running container
    load      Load an image from a tar archive
    login     Register or Login to the docker registry server
    logs      Fetch the logs of a container
    port      Lookup the public-facing port which is NAT-ed to PRIVATE_PORT
    ps        List containers
    pull      Pull an image or a repository from the docker registry server
    push      Push an image or a repository to the docker registry server
    restart   Restart a running container
    rm        Remove one or more containers
    rmi       Remove one or more images
    run       Run a command in a new container
    save      Save an image to a tar archive
    search    Search for an image in the docker index
    start     Start a stopped container
    stop      Stop a running container
    tag       Tag an image into a repository
    top       Lookup the running processes of a container
    version   Show the docker version information
    wait      Block until a container stops, then print its exit code
```

CLI by Example

Rather than just run through each command it would be more informative to continue the example that was given in the dockerfile configuraton section earlier. We will take that configuration through to running the container on a host and investigating what we can do with the CLI to manage the lifecycle of the container.

Building an Image

The first command option to explore is *build*.

```
$ docker help build

Usage: docker build [OPTIONS] PATH | URL | -

Build a new container image from the source code at PATH

  --no-cache=false: Do not use cache when building the image
  -q, --quiet=false: Suppress the verbose output generated by the containers
  --rm=true: Remove intermediate containers after a successful build
  -t, --tag="": Repository name (and optionally a tag) to be applied to the resulting image
```

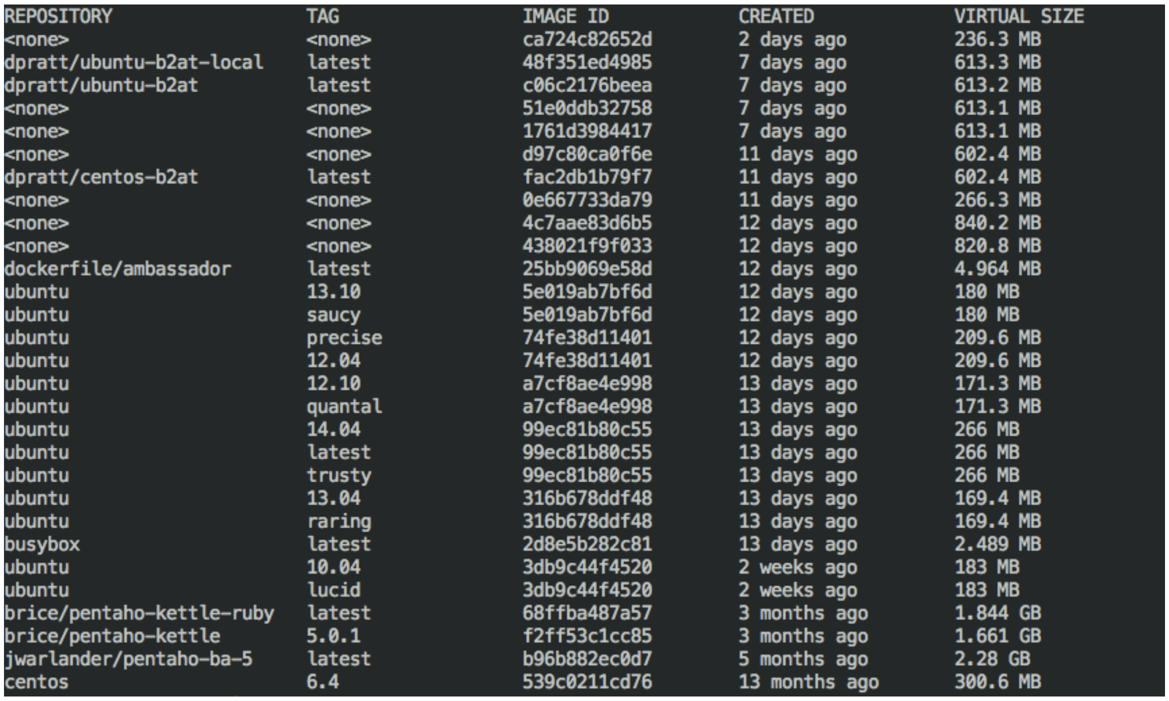
The build command takes the dockerfile configuration and creates an image that is stored for usage on the local machine. The two options to truly note here are the –no-cache option which you might want to use to verify that you are picking up any changes from your base images and the -t option should always be used to give your resulting an image a user friendly name for reference when managing it. In order to build our example from earlier (node image from the configuration secton), we run our command from the directory containing *dockerconfig* file. The syntax is as follows.

```
# build our image with the tag name of dpratt/unbuntu-nodejs
docker build -t dpratt/ubuntu-nodejs .
```

You can use whichever name you like for your tag but the general idea is that if you are going to publish the image to the docker index, then the tag would be of the form *githubusername/*imagename. Note that we are not using *sudo* to run the command. As discussed earlier, if our user belongs to the *docker* group then we can run without sudo privileges.

Verify the Image

Following the previous step we would want to verify that our new image is present in our local list. We can use the *images* command to do this. Issuing *docker images* will result in a list of the images which have been created or pulled to the local machine.



```
REPOSITORY                TAG            IMAGE ID         CREATED          VIRTUAL SIZE
<none>                    <none>         ca724c82652d     2 days ago       236.3 MB
dpratt/ubuntu-b2at-local  latest         48f351ed4985     7 days ago       613.3 MB
dpratt/ubuntu-b2at        latest         c06c2176beea     7 days ago       613.2 MB
<none>                    <none>         51e0ddb32758     7 days ago       613.1 MB
<none>                    <none>         1761d3984417     7 days ago       613.1 MB
<none>                    <none>         d97c80ca0f6e     11 days ago      602.4 MB
dpratt/centos-b2at        latest         fac2db1b79f7     11 days ago      602.4 MB
<none>                    <none>         0e667733da79     11 days ago      266.3 MB
<none>                    <none>         4c7aae83d6b5     12 days ago      840.2 MB
<none>                    <none>         438021f9f033     12 days ago      820.8 MB
dockerfile/ambassador     latest         25bb9069e58d     12 days ago      4.964 MB
ubuntu                    13.10          5e019ab7bf6d     12 days ago      180 MB
ubuntu                    saucy          5e019ab7bf6d     12 days ago      180 MB
ubuntu                    precise        74fe38d11401     12 days ago      209.6 MB
ubuntu                    12.04          74fe38d11401     12 days ago      209.6 MB
ubuntu                    12.10          a7cf8ae4e998     13 days ago      171.3 MB
ubuntu                    quantal        a7cf8ae4e998     13 days ago      171.3 MB
ubuntu                    14.04          99ec81b80c55     13 days ago      266 MB
ubuntu                    latest         99ec81b80c55     13 days ago      266 MB
ubuntu                    trusty         99ec81b80c55     13 days ago      266 MB
ubuntu                    13.04          316b678ddf48     13 days ago      169.4 MB
ubuntu                    raring         316b678ddf48     13 days ago      169.4 MB
busybox                   latest         2d8e5b282c81     13 days ago      2.489 MB
ubuntu                    10.04          3db9c44f4520     2 weeks ago      183 MB
ubuntu                    lucid          3db9c44f4520     2 weeks ago      183 MB
brice/pentaho-kettle-ruby latest         68ffba487a57     3 months ago     1.844 GB
brice/pentaho-kettle      5.0.1          f2ff53c1cc85     3 months ago     1.661 GB
jwarlander/pentaho-ba-5   latest         b96b882ec0d7     5 months ago     2.28 GB
centos                    6.4            539c0211cd76     13 months ago    300.6 MB
```

docker images output

If all went well we should see the name of our new image in the list.

Running a Container

Now that we have our image created and it shows in our list output, we are good to start a container based upon this image. The *run* command is used to startup a container based upon a custom image or base image of our choosing. The command syntax is more complex than the previously seen commands. It takes the general following form.

```
Usage: docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

Run a command in a new container

  -a, --attach=[]: Attach to stdin, stdout or stderr.
  -c, --cpu-shares=0: CPU shares (relative weight)
  --cidfile="": Write the container ID to the file
  -d, --detach=false: Detached mode: Run container in the background, print new container id
  -e, --env=[]: Set environment variables
  --entrypoint="": Overwrite the default entrypoint of the image
  --env-file=[]: Read in a line delimited file of ENV variables
  --expose=[]: Expose a port from the container without publishing it to your host
  -i, --interactive=false: Keep stdin open even if not attached
  --link=[]: Add link to another container (name:alias)
  -m, --memory="": Memory limit (format: <number><optional unit>, where unit = b, k, m or g)
  --name="": Assign a name to the container
  -p, --publish=[]: Publish a container's port to the host
               (format: ip:hostPort:containerPort |
                              ip::containerPort | hostPort:containerPort)
                              (use 'docker port' to see the actual mapping)
  -P, --publish-all=false: Publish all exposed ports to the host interfaces
  --privileged=false: Give extended privileges to this container
  --rm=false: Automatically remove the container when it exits (incompatible with -d)
  -u, --user="": Username or UID
  -v, --volume=[]: Bind mount a volume (e.g. from the host:
                     -v /host:/container, from docker: -v /container)
  --volumes-from=[]: Mount volumes from the specified container(s)
  -w, --workdir="": Working directory inside the container
```

The command has a few more options revolving around LXC and DNS but those are best read in the documentation. I haven’t had a need yet to use those options, so for one to experiment with Docker it’s probably not necessary.

Using our image from before, we can run the following command to run our container based on this image.

```
docker run -p 49160:3000 -d dpratt/ubuntu-nodejs
```

With this command we are instantiating our container ***dpratt/ubuntu-nodejs*** and the **-p** option is telling the docker daemon to expose the container port 3000 to the host port 49160. This port mapping creates the network bridge from the host to the container so we can access the container service on port 49160 after it is running. The dockerfile configuration from earlier exposed the port 3000 from the container, so the **-p** option here just maps that exposed port to a host port of our choosing. The **-d** options instructs docker to run the container as a detached container. This means that we basically want to run the container in the background. This would be similar to running a command as ***nohup somescript.sh &*** in unix.

Verification of Container State

If the container was started with the **-d** (detached) option, then the shell will not show the logs as outputed by the command that is being run by the container. This is not an issue as docker provides a few methods to handle this.

The commands that are used to inspect a running container use the containers id that is generated when the container is started. The simplest way of doing this is to capture the ID upon running the command. The following works to make the container id more referenceable for future commands.

```
CONTAINER_ID=`docker run -d -p 49160:3000 dpratt/ubuntu-nodejs`
```

This will capture our container id in the environmental variable for use in later commands. If you do not capture the id, then you can run the docker ps command to list the id’s of each active container. The ID can be groked from this output and used later.

Running the *docker ps* command now should give us the following output.

```
$ docker ps
CONTAINER ID        IMAGE                           COMMAND                 CREATED             S
1cf68b9bdb50        dpratt/ubuntu-nodejs:latest     node /src/server/ser    8 minutes ago
```

The output from the docker ps command shows the container id which can be used to attach to the running container or otherwise control its lifecycle. It is important to notice that the name is not null here. We did not specify a *–name* option on the run command so docker has generated a random name for the container. When we look at linking containers next, this name becomes more important.

Container Linking

One core of idea of docker is that each container is really only running one process. If you were to build out a typical web application, you might want to create a containers to hold nginx, redis, nodejs, couchbase etc. With these containers created at the process level, the issue becomes visibility between containers at run time. Fortunately, docker solves this problem easily with the concept of container naming and linking. As we detailed above, the *–name* option gives the newly created container a referenceable name. This name can be used for any of the docker commands but more importantly the name gives a container a method to interact with another running container. Linking containers is accomplished with the *–link=[]* command option. The option can take a list of *name:alias* pairs.

```
sudo docker run -t -i --link redis:db --name webapp ubuntu bash
```

This command will establish a link to the redis container from this new webapp container with the alias db. The interesting part of this is that the environmental information from the linked container is made available to the child container. The child container now has access to the parent' environmental configuration, so the container can use this information to connect to that parent container's resources. The configuration data is prefixed with the name of the parent container, such that a variable in the above parent container named TCP_PORT would be available as DB_TCP_PORT in the child container.

In the case of a database, the parent container could simply expose the connection information in the environment, and this could be used by the child to make the connection without being explicitly configured from the dockerfile or runtime to with this data.

Conclusion

This paper has covered in detail the basic usage of Docker to create micro containers and why one might want to do so. Docker is most certainly a new technology, but is based almost entirely upon technology from the underlying linux kernel which has been in existence for some time. As detailed in the paper, the benefits of the speed of building and starting containers, makes Docker a very interesting technology when compared to existing solutions such as VMWare. As applications have moved more and more toward the concept of separation of concerns within the application architecture, Docker should allow organizations to move their infrastructure toward the concept of SOC with hardware virtualization at the micro container level.