Docker procedures for RoboCup MSL

Peter van Lith – V0.0 – 18-11-17

V0.1 – 10-12-17

# About this document

This document is a gentle introduction to Docker and describes how to install Docker and then create an image. It is written to support teams that want to participate in the shared simulator and explains how to create a Dockerized image for their team’s software.

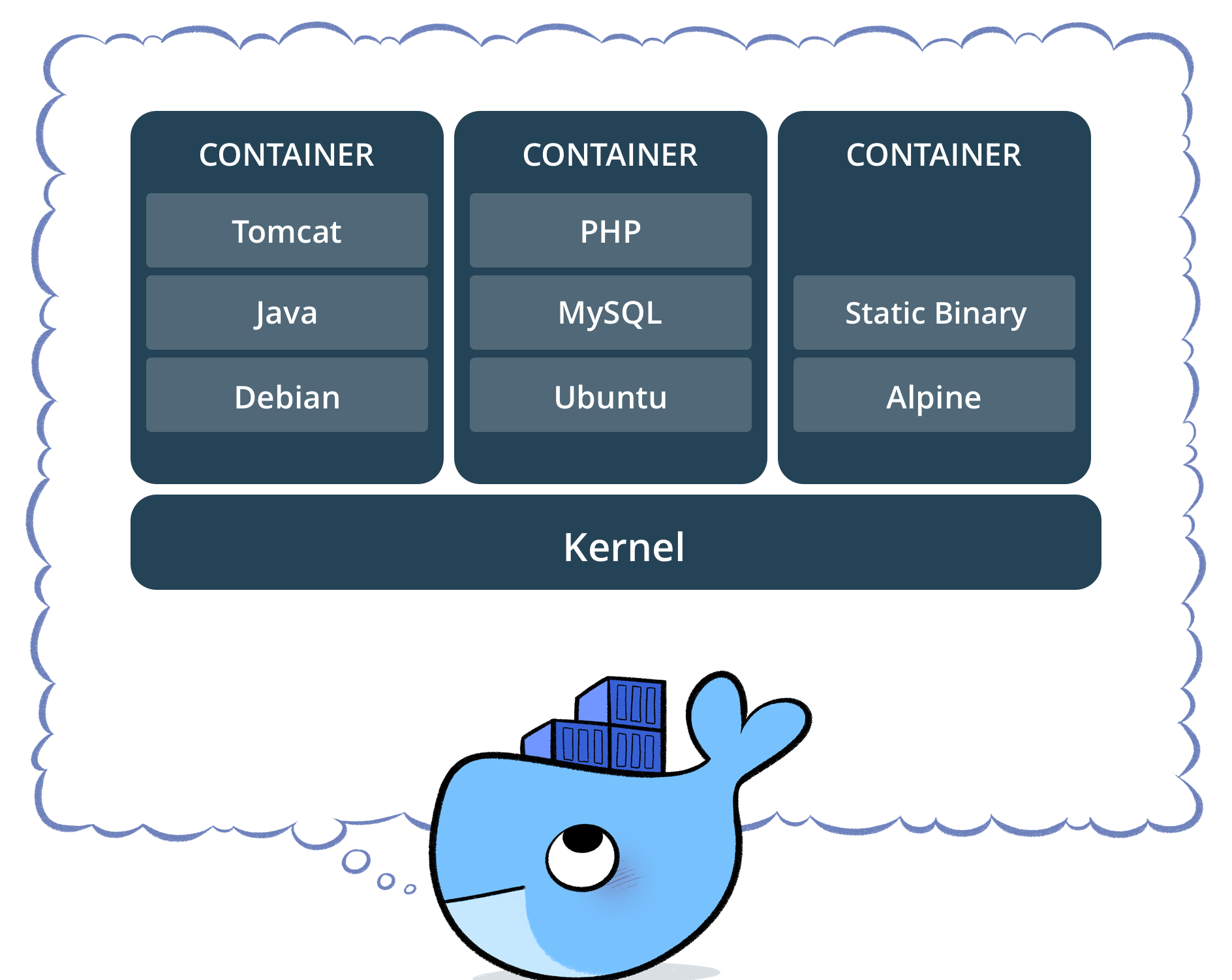
This is only a first draft, intended to set up and test the required procedures, gradually growing into the full document.

# General Docker procedures

## Why Docker

Docker is a container manager that allows the creation of a stand-alone executable package of a piece of software that includes everything to run it independent of the current installation. It contains everything needed like code, system tools, libraries and data and is available for Linux, Mac and Windows.

For the MSL simulator it was decided to use Docker to distribute the simulator and the software for every team, participating in the MSL simulation project.



There will be a number of containers, the main one will be the MSL simulator itself, needed to run the simulator with a number of standard models. A model is a description of a robot team, its robots and the strategy part of the software.

The second part is a container for every team participating that contains the specific robot model and the strategy layer of the team software. To maintain the confidentiality of the team software it will only contain the models and the runtime of the software that allows the simulator to let any team play against any other team.

## Installing Docker

Installation of Docker is described in detail in the following documents. The steps required to install are as described below. Please check the links for details on installing.

We will be using the Community Edition (CE) of Docker to allow a small footprint and keep the download times manageable.

<https://www.docker.com/community-edition#/download>

For Windows and Mac there is a simple installation program that will install the software. For Linux the process is a little more involved. First the library must be installed and includes the following steps as described here:

<https://docs.docker.com/engine/installation/linux/docker-ce/ubuntu/#install-using-the-repository>

1. Set up the Repository
   1. Update the package index
   2. Install repository packages
   3. Add Docker GPG key
   4. Set up the repository
2. Install the Docker CE packages
   1. Update the package index
   2. Install Docker CE
   3. Verify Docker CE installation
3. Perform post-installation steps

For Linux we standardize on the 16.04 stable releases, so all users share the same environment.

## Additional installation options

Apart from the standard ways of installing Docker there is also the get-docker convenience script as follows:

curl -fsSL get.docker.com -o get-docker.sh

sudo sh get-docker.sh

If you add your user to the "docker" group allows you can use docker commands without sudo:

sudo usermod -aG docker ${USER}

## Creating a simple Docker container

Create an empty directory. Change directories (cd) into the new directory, create a file called Dockerfile, please note the Capital D. Copy-and-paste the following content into that file, and save it. Take note of the comments that explain each statement in your new Dockerfile.

# Use an official Python runtime as a parent image

FROM python:2.7-slim

# Set the working directory to /app

WORKDIR /app

# Copy the current directory contents into the container at /app

ADD . /app

# Install any needed packages specified in requirements.txt

RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container

EXPOSE 80

# Define environment variable

ENV NAME World

# Run app.py when the container launches

CMD ["python", "app.py"]

Now create two more files, requirements.txt and app.py with the following contents:

### requirements.txt

Flask

Redis

### app.py

from flask import Flask

from redis import Redis, RedisError

import os

import socket

# Connect to Redis

redis = Redis(host="redis", db=0, socket\_connect\_timeout=2, socket\_timeout=2)

app = Flask(\_\_name\_\_)

@app.route("/")

def hello():

try:

visits = redis.incr("counter")

except RedisError:

visits = "<i>cannot connect to Redis, counter disabled</i>"

html = "<h3>Hello {name}!</h3>" \

"<b>Hostname:</b> {hostname}<br/>" \

"<b>Visits:</b> {visits}"

return html.format(name=os.getenv("NAME", "world"), hostname=socket.gethostname(), visits=visits)

if \_\_name\_\_ == "\_\_main\_\_":

app.run(host='0.0.0.0', port=80)

## Dependencies

Every program has a number of dependencies. As can be seen in the small example script all dependencies need to be included in the dockerfile.

While we were setting up an example docker version of the current Tech United Simulator we encountered some rather curious dependencies that could also play a role in the applications of the other teams. So this is a warning to be careful when setting up the Docker version of you team’s client version. Here we describe the Tech United dependencies as an example for other teams to check before planning the creation of you MSL simulator client.

The main body of the Tech United Turtle software is based on MatLab with additional software mostly written in C++. So all MatLab dependencies need to be included in the Docker image. Fortunately MatLab allows royalty-free distribution of the run-time libraries. Other software might not have such royalty-free libraries and might need additional licensing facilities. Because this imposes severe limitations, we would suggest to avoid using such software in the client part of the simulator.

A more disturbing problem was a dependency that was not so obvious. To make sure that during a competition all team members could have restricted access to the Turtle software, every computer is given a unique identifier in the form of a dedicated computer name. The Turtle simulator currently has the requirement that only computers with this dedicated name can start up the softeware.

Because the client part of the MSL simulator must include the strategic layer of the Turtle software, this should not only the MatLab runtime modules but must also adhere to the computer naming convention. This however is not possible in a Docker image. So we will need to change the setup of the Turtle software to remove this dependency. Other teams might have similar dependencies that are hardware related and cannot be implemented in a Docker image.

So please be aware of this and check your software for such bottlenecks. There might be other things that we currently have not thought about that may be of influence for the implementation of the MSL simulator.

## Build the app

We are ready to build the app. Make sure you are still at the top level of your new directory. Here’s what ls should show:

$ ls

Dockerfile app.py requirements.txt

Now run the build command. This creates a Docker image, which we’re going to tag using -t so it has a friendly name.

docker build -t hello-simul .

You will see the following output:

Sending build context to Docker daemon 4.608kB

Step 1/7 : FROM python:2.7-slim

2.7-slim: Pulling from library/python

d13d02fa248d: Pull complete

a2c103c31b60: Pull complete

33bfff8f2f5e: Pull complete

5b66f3cbc9f3: Pull complete

Digest: sha256:1f9abab4af336c05eb13091aa7d704b8044090e022081d950bb507a801ca378a

Status: Downloaded newer image for python:2.7-slim

---> b0259cf63993

Step 2/7 : WORKDIR /app

---> bb6e23dd08f9

Removing intermediate container 8bafbd2159a6

Step 3/7 : ADD . /app

---> 2ac8cc4e64f5

Step 4/7 : RUN pip install --trusted-host pypi.python.org -r requirements.txt

---> Running in 8c35c928b489

Collecting Flask (from -r requirements.txt (line 1))

Downloading Flask-0.12.2-py2.py3-none-any.whl (83kB)

Collecting Redis (from -r requirements.txt (line 2))

Downloading redis-2.10.6-py2.py3-none-any.whl (64kB)

Collecting itsdangerous>=0.21 (from Flask->-r requirements.txt (line 1))

Downloading itsdangerous-0.24.tar.gz (46kB)

Collecting Jinja2>=2.4 (from Flask->-r requirements.txt (line 1))

Downloading Jinja2-2.10-py2.py3-none-any.whl (126kB)

Collecting Werkzeug>=0.7 (from Flask->-r requirements.txt (line 1))

Downloading Werkzeug-0.13-py2.py3-none-any.whl (311kB)

Collecting click>=2.0 (from Flask->-r requirements.txt (line 1))

Downloading click-6.7-py2.py3-none-any.whl (71kB)

Collecting MarkupSafe>=0.23 (from Jinja2>=2.4->Flask->-r requirements.txt (line 1))

Downloading MarkupSafe-1.0.tar.gz

Building wheels for collected packages: itsdangerous, MarkupSafe

Running setup.py bdist\_wheel for itsdangerous: started

Running setup.py bdist\_wheel for itsdangerous: finished with status 'done'

Stored in directory: /root/.cache/pip/wheels/fc/a8/66/24d655233c757e178d45dea2de22a04c6d92766abfb741129a

Running setup.py bdist\_wheel for MarkupSafe: started

Running setup.py bdist\_wheel for MarkupSafe: finished with status 'done'

Stored in directory: /root/.cache/pip/wheels/88/a7/30/e39a54a87bcbe25308fa3ca64e8ddc75d9b3e5afa21ee32d57

Successfully built itsdangerous MarkupSafe

Installing collected packages: itsdangerous, MarkupSafe, Jinja2, Werkzeug, click, Flask, Redis

Successfully installed Flask-0.12.2 Jinja2-2.10 MarkupSafe-1.0 Redis-2.10.6 Werkzeug-0.13 click-6.7 itsdangerous-0.24

---> 6c8d39508d04

Removing intermediate container 8c35c928b489

Step 5/7 : EXPOSE 80

---> Running in 4d005281af11

---> 1703bb54cdff

Removing intermediate container 4d005281af11

Step 6/7 : ENV NAME Simulator

---> Running in a9734cee308d

---> 4c794de5ec8f

Removing intermediate container a9734cee308d

Step 7/7 : CMD python app.py

---> Running in b0b65aa27f59

---> 8210ce862030

Removing intermediate container b0b65aa27f59

Successfully built 8210ce862030

Successfully tagged hello-simul:latest

Where is your built image? It’s in your machine’s local Docker image registry:

$ docker image ls

REPOSITORY TAG IMAGE ID

hello-simul latest 326387cea398

You may remove the image with the command:

docker image rm 326387cea398

## Run the app

Run the app, mapping your machine’s port 4000 to the container’s published port 80 using -p:

docker run -p 4000:80 hellosimul

You should see a message that Python is serving your app at http://0.0.0.0:80. But that message is coming from inside the container, which doesn’t know you mapped port 80 of that container to 4000, making the correct URL

http://localhost:4000.

Go to that URL in a web browser to see the display content served up on a web page, including “Hello Simulator” text, the container ID, and an error message that.

Now let’s run the app in the background, in detached mode:

docker run -d -p 4000:80 simulatorhello

You get the long container ID for your app and then are kicked back to your terminal. Your container is running in the background. You can also see the abbreviated container ID with docker container ls (and both work interchangeably when running commands):

$ docker container ls

CONTAINER ID IMAGE COMMAND CREATED 1fa4ab2cf395 hellosimul "python app.py" 28 seconds ago

You’ll see that CONTAINER ID matches what’s on http://localhost:4000.

Now use docker container stop to end the process, using the CONTAINER ID, like so:

docker container stop 1fa4ab2cf395

## Sharing the image

*Note: This part needs further details on how to put each team’s Docker image in the central repository. It is a general layout of the required contents.*

### Log into GitHub

### Tag the image

The notation for associating a local image with a repository on a registry is username/repository:tag. The tag is optional, but recommended, since it is the mechanism that registries use to give Docker images a version. Give the repository and tag meaningful names for the context, such as team-xx:V0.0. This will put the image in the team-xx repository and tag it as V0.0.

Now, put it all together to tag the image. Run docker tag image with your username, repository, and tag names so that the image will upload to gitHubtion. The syntax of the command is:

docker tag image username/repository:tag

For example:

docker tag simul-model xx/team-xx:v0.0

Run [docker images](https://docs.docker.com/engine/reference/commandline/images/) to see your newly tagged image. (You can also use docker image ls.)

$ docker images REPOSITORY TAG IMAGE ID CREATED SIZE friendlyhello latest d9e555c53008 3 minutes ago 195MB xx/team-xx v0.0 d9e555c53008 3 minutes ago 195MB python 2.7-slim 1c7128a655f6 5 days ago 183MB ...

### Publish the image

Upload your tagged image to the repository:

### Pull and run the image from the remote repository

From now on, you can use docker run and run your app on any machine with this command:

docker run -p 4000:80 username/repository:tag

If the image isn’t available locally on the machine, Docker will pull it from the repository.

## Recap and cheat sheet

Here is a list of the basic Docker commands from this page, and some related ones if you’d like to explore a bit before moving on.

docker build -t friendlyname . # Create image using this directory's Dockerfile docker run -p 4000:80 friendlyname # Run "friendlyname" mapping port 4000 to 80 docker run -d -p 4000:80 friendlyname # Same thing, but in detached mode docker container ls # List all running containers docker container ls -a # List all containers, even those not running docker container stop <hash> # Gracefully stop the specified container docker container kill <hash> # Force shutdown of the specified container docker container rm <hash> # Remove specified container from this machine docker container rm $(docker container ls -a -q) # Remove all containers docker image ls -a # List all images on this machine docker image rm <image id> # Remove specified image from this machine docker image rm $(docker image ls -a -q) # Remove all images from this machine docker login # Log in this CLI session using your Docker credentials docker tag <image> username/repository:tag # Tag <image> for upload to registry docker push username/repository:tag # Upload tagged image to registry docker run username/repository:tag # Run image from a registry

# The Simulator procedures

This section describes the procedures to create the team container and using it with the simulator.

# The Simulator container

The actual simulator will be the server that runs the simulation. It will run on any machine that has the simulator installed and will also contain a number of default clients with standard models. This basic configuration will run the simulator with two clients. The use of clients will be configurable. Clients may be local or remote.

# Creating the Team container

The team container will be created using the procedures, described in the first part of this document. It will be distributed using the Github repository and can be retrieved from there by all participating teams.

# Using the Team container

There will be a number of client containers, at least one for each participating team. In the future there may be several versions with possibly different models. We need a way to describe the various.

A client container may run on the same machine as the simulator or on another machine connected through the network or even remotely.

With the client containers any team will be able to play a simulation of it’s own team against any of the participating teams.