Lightweight ROS Development Environments Using Docker

Jack Bradach <jack@bradach.net> 2016/08/23

Introduction / Problem Statement:

ROS releases (eg, Indigo, Jade, Kinetic) are tied to Ubuntu releases (Trusty, Vivid, Xenial). Not all parts of ROS are immediately available (Turtlebot, for example, is not yet available in Kinetic). This presents an annoyance for development and deployment: you have to install, configure, and maintain a crusty version of Ubuntu around to test your code on.

One alternative is to use Docker (http://www.docker.io/) to create a ROS environment inside a software container. Read up on it when you get a chance; it's what all the cool kids today are using for virtualization. For now you can treat them somewhere between a virtual machine and a chroot jail. From the point of view of ROS (eg, Indigo) running in the container, it's sitting on a pristine Ubuntu 14.04 LTS install. The host can be a modern version of Ubuntu or any flavor of Linux you like. Rumor has it that OS X and Windows can host Linux containers as well, although we will not be covering that in this paper.

This is a guide to get a ROS-in-Docker environment set up, using the Programming Robots With ROS examples. The provided Dockerfile will download an Ubuntu Trusty image, install ROS Indigo, and check out the latest copy of the rosbook repo to the container. From there, you can launch one or more (yes, simultaneously) instances of a ROS container to play with.

Getting Started (install Docker):

I'm going to assume that you are running Ubuntu 16.04 (or later) and have sudo/root access. If you're on a different flavor of Linux, consult the official Docker documentation or type "How do I install Docker on <McGuffin> Linux" into Google. No, seriously; the official documentation is really good. They have step-by-step instructions for most of the common flavors. For off-brand Linuxes, someone out there has likely done a write-up; learn from them!

Go to https://docs.docker.com/engine/installation/linux/ubuntulinux/ and run through the required steps to install the official Docker packages. You should also do the "Create a docker group" optional step, as this will allow you to interact with Docker as your regular user. You'll also probably want Docker to start on boot, so do that one as well. Reboot and make sure the Docker "Hello World" still works.

Building the ROS Container:

At this point Docker is installed, yes? Good. Now you'll want to create a new directory somewhere and copy the rosbook Dockerfile to it.

```
$ mkdir -p ~/workspace/rosbook
$ cd ~/workspace/rosbook
<copy Dockerfile to .>
```

This next step is going to kick off a whole lot of business. The required components for the image will be downloaded and the whole mess will be glued together in ways that give me a headache to even think about; It's black magic, for sure. Ready?

```
$ docker build --rm -t rosbook .
```

Note the period argument at the end; it's important. I'm not gonna say why, you just make sure to include it. You've got time to go get a coffee. No, not office coffee; that place up the street. You just installed Docker and kicked off your second container build already. You've earned it!

And we're back! Wait, what did I just do?

You told Docker to build a new container and name it 'rosbook.' The -t (short for --tag) parameter sets the name the final container will be known by. When you run a container, you can either give it a very long SHA-1 hash identifier or a short friendly name. The --rm tells it to remove the intermediate containers it downloads or otherwise creates in the process of building a Dockerfile. If you are developing or modifying an existing Dockerfile, leave this off so that it doesn't have to rebuild everything from scratch if you make a change. The period provides the "context" for the build. Everything under that directory will get copied into the container.

Next, some of the interesting bits of the Dockerfile. The "FROM ros:indigo-ros-base" line tells Docker to base our image on the ROS Indigo image. If you wanted to use a different version of ROS in your container it is as simple (for some values of simple) as changing 'indigo' to, for example, 'kinetic' and rebuilding the container.

The script creates a catkin workspace (at /catkin_ws/src) and checks out the latest version of the rosbook repository from Github. The Gazebo model repository is also cloned; otherwise it'll download them every time that Gazebo is spawned. Xeyes is included because it's an easy way to check that your container is able to paint on your desktop.

But what about my fancy graphics chipset? VMs don't normally directly touch the hardware, do they? Software rendered 3D is pretty dang terrible.

Making sure that you have hardware accelerated video is the whole reason for this tutorial! Remember that we're running Linux in Linux. Userspace programs don't get to touch the hardware, right? They'll usually use libraries which use either device nodes (/dev/dri) or shared memory regions to call into the driver. Docker let's us map parts of the host into the container, and we can do just that with the nodes needed to talk to the video card and X server. As you can probably guess from the length of the rest of this section, some assembly may be required.

The amount of extra effort needed to make this a thing that happens varies based on your graphics chipset (Nvidia, Intel, ATI). If you have a recent (Skylake or later) Intel GPU, you mostly just need to make sure the latest OpenGL stack is installed in the guest; this should be covered by the Dockerfile that should have accompanied this document. NVidia gets slightly more complicated. Due to the way their GPU architecture works, a special wrapper is needed around Docker to correctly work as intended. They go into it in great detail on their FAQ, actually, and it's an interesting read. Their documentation is also solid and provides a few cut-and-pasted lines to install nvidia-docker. With the caveat that you really ought to go check https://github.com/NVIDIA/nvidia-docker for updates, here is the magic phrase that pays as of the time of this writing:

```
# Install nvidia-docker and nvidia-docker-plugin
$ wget -P /tmp
https://github.com/NVIDIA/nvidia-docker/releases/download/v1.0.0-rc.3/nvidia-docker_1.0.0.rc.3-1_amd64.deb && \
sudo dpkg -i /tmp/nvidia-docker*.deb && \
rm /tmp/nvidia-docker*.deb

# Test nvidia-smi; this makes sure that nvidia-docker has hardware
acceleration.
$ nvidia-docker run --rm nvidia/cuda nvidia-smi
```

There are also a couple of lines in the Dockerfile that are taken from the NVidia page; it's doing some sort of overlay so that the container automatically gets a usable driver. They'll be ignored on non-NVidia containers.

ATI has no data on the ROS Docker wiki. This may indicate that no special snowflake handling is needed and it "just works" with no hours of maddening debug necessary. It is a happy world. Unfortunately, as I only have NVidia and Intel to test on, you'll have to do some Google research if the examples don't work.

Running the newly minted container:

First we have to disable X access control so the container can write to the host's X server. We'll tell it that all connections originating locally can access. This should be relatively safe on a single-user machine.

```
$ xhost +local:root
```

Next we're going to run the actual container. Since this is the first time we're running the container, we're going to type out the whole hairy thing. Later, we'll wrap it in an alias.

```
$ docker run -it --rm \
    --ipc host \
    --volume=/tmp/.X11-unix:/tmp/.X11-unix \
    --device=/dev/dri:/dev/dri \
    --env="DISPLAY" \
    rosbook \
    /bin/bash
```

If you have an Nvidia chipset, you'll need to substitute 'nvidia-docker' for docker, but the rest should be the same.

You'll now be in a bash shell in /catkin_ws. This is the container's filesystem. I recommend making sure that OpenGL works correctly by running glxgears. If that worked, you can kick off a Gazebo Turtlebot with roslaunch:

```
# roslaunch turtlebot_gazebo turtlebot_world.launch
```

After some whirring, you'll be looking at Virtual Turtlebot, as it silently ponders existence. We'll now join these regularly scheduled programs, already in progress.

For as long as Virtual Turtlebot could remember, there had only ever been the six of them. Six in the whole world. None could remember how, specifically, they came to be the sole inhabitants of a large, featureless grey plane nor for what purpose they had been instantiated. They simply were and would continue to be, as their experiences up until this point taught them to expect. It was a good, simple existence with no excitement or drama. VT tried to enjoy himself and fit in with the other immobiles, but he could feel them always staring at him. More specifically at his wheels. He'd heard the whispers, "why does he bother with having wheels if he never goes anywhere?" VT would counter that he didn't see any of them attempting to translate themselves either, but deep down he knew that he simply had no place to go and nothing to motivate him to get there. And so they persisted for untold cycles, admiring each others specular highlights and the enjoying the flicker of anti-aliasing errors in their shadows.

Virtual Turtlebot doesn't have anything sending Twist messages to the topic that it's expecting to get teleop messages on. Without these, it's more of a Slackerbot and can't be relied upon to do anything. The Wanderbot example from the book can generate theswhen our environment is containerized? There are a couple of options.

Since the Wanderbot is already in the image, we can attach to the running container from another terminal and run it. First we need the name of the container (or it's id). You can find this with:

```
$ docker ps
```

It'll spit out a container hash ID and show the container names. It normally makes up its own random name per container instance, but you can override this with --name when you run a container. While we could start a shell and run by hand as before:

```
$ docker exec -it <container_name> /bin/bash
```

We could also pull out the aces and run the Wanderbot as a one-liner. I've split it up so it's hopefully easier to follow what we're doing:

```
$ docker exec -it rosbook_test \
      /bin/bash -c 'source devel/setup.bash && \
      src/rosbook/wanderbot/wander.py cmd_vel:=cmd_vel_mux/input/teleop'
```

This will cause the Virtual Turtlebot to start wandering around. It'll bounce off the existing models before heading off into the "sunset."

Virtual Turtlebot struggled finding his bearings through the haze of digital eternity. An unseen point sun beat down from overhead, lighting all things evenly and making distance impossible to estimate visually. He'd gone too far; past the grid and the grey and into the wild lighter shade of grey. There's nothing out here and there was never meant to be. Worse, VT no longer had any solid reference for where "home" was. Its short range sensors were useless out here. Accumulated rounding errors in its dead reckoning made his estimation of where The Grid was no better than a simple guess. It was screwed, plain and simple. The day had started off so well, otherwise.

Switch back to the original terminal you started Docker in (it should have messages from Gazebo being displayed). Now hit control-C to send a sigint and get it to shut down ROS. This will also cause the other terminal controlling Virtual Turtlebot to exit as well, as the container will be stopped.

VT didn't even know why it continued on. Something kept driving it forward, always pushing a direction or a turn. There was a pattern to be followed, but he could discern no meaning behind it. Abruptly, the messages stopped. Virtual Turtlebot, now motionless, didn't know where the relentless Twisting voice has gone, but it was thankful for the rest. A moment later, the world flickered. Staring in horror at the sky, VT heard a rising call that froze its state machine cold. He'd heard it many times, but never so overwhelmingly everywhere. The sound of the global destructor tore across the world, de-constructing and freeing reality as it went. Viscous red squiggles screamed across the plain towards VT. He barely had time to register them wrapping around his primitives before his resources were freed.

OH WAIT, Let's talk about ephemeralism:
Data in Docker containers is not persistent. When you stop the container, all changes are discarded. You get a brand new unopened instance when you start a new copy. This would be very bad for development; all your work would go up in smoke as soon as you closed Docker. There's an easy fix, of course, and that's:

Mounting external volumes (like your work or logging directories!)
Docker lets you mount folders from the host filesystem into the container using the volume parameter. You've seen this already, it's how we're sharing the .X11 session info. The volume is directly mapped to the container through some filesystem voodoo which I believe to be UnionFS. You'll want to map in a directory from the host to use as a catkin_workspace and a couple directories for logs from ROS and Gazebo. Here's an example you can modify to fit your needs:

```
$ docker run -it --rm \
    --ipc host \
    --volume=/tmp/.X11-unix:/tmp/.X11-unix \
    --device=/dev/dri:/dev/dri \
    --env="DISPLAY" \
    --volume=/home/user/ros/catkin_ws:/catkin_ws \
    --volume=/home/user/ros/.gazebo:/root/.gazebo \
    --volume=/home/user/ros/.ros:/root/.ros \
    rosbook \
    /bin/bash
```

You can use as many volumes as you want. Just remember that if you are saving to someplace in the container that is not a volume, it's going to disappear forever when you stop the container.

Shortening the Docker commands with Aliases

If you plan on running a Docker container with the same configuration over and over again, it's not a bad idea to make an alias for it. Otherwise you'll be gnarling your fingers with how long a docker run command can be. Here's one that will behave like roslaunch, only within a container. Substitute your own directories. Make sure to substitute nvidia-docker if you need to!

```
$ alias d_roslaunch=\
'docker run -it --rm \
    --ipc host \
    --volume=/tmp/.X11-unix:/tmp/.X11-unix \
    --device=/dev/dri:/dev/dri \
    --env="DISPLAY" \
    --volume=${HOME}/ros/catkin_ws:/catkin_ws \
    --volume=${HOME}/ros/.gazebo:/root/.gazebo \
    --volume=${HOME}/ros/.ros:/root/.ros \
    rosbook \
    roslaunch'
```

This alias, which you can put at the end of your .bashrc or `simply` cut and paste into a terminal, allows you to easily "roslaunch" a container:

```
$ d_roslaunch turtlebot_gazebo turtlebot_world.launch
```

*VT faded back into realization that he was still a being. Unable to see, he tried to call out but realized that he had no… anything, really. Suddenly a buzzing and snap. A stream of data from his restored sensors. He opened his eyes to a Constructor flitting about with a high pitched but not unpleasant hum. It was quickly and efficiently assembling him, mostly from what appeared to be a pile of used primitives and memory blocks. Assembling… he was being re-constructed? The horror of his situation crept in around the edges; Yes, *a* VT was being constructed. He was simply a leftover bit of the last world that happened to also fit into the new one. The Constructor finished instantiating the rest of his model before resetting the VT back to a known initial state. Virtual Turtlebot didn't mind, though, because as long as he could remember, there had only ever been the six of them. Six in the whole world.*

Cleaning up unused containers and images

Docker saves a lot of intermediary data when it's building images. That way, if you make a change to a Dockerfile, it doesn't have to download everything from scratch. Once your image is stable, you can clean up these untagged images with this one liner:

```
docker rm $(docker ps -qa --no-trunc --filter "status=exited")
```

```
docker rmi $(docker images --filter "dangling=true" -q --no-trunc)
```

Non-Linux Docker Hosts

Docker in Windows is easy to install, although it does have Hyper-V as a requirement. This may matter to you if you run Virtualbox, as the method the two hypervisors use for… hypervising are mutually exclusive. The reason why is neat; go Google it!

Put your Dockerfile someplace convenient and from Powershell you can run the docker build command as in Linux. It'll build the container and you can run it using the same methods described above. The catch is that there doesn't seem to be support yet for video acceleration passthrough. If you want to run Windows and can get by without Gazebo, this may be an option to consider.

Summary and extra notes

And we're at the end! To recap, here's what we covered:

1) The concept of using Docker to create create lightweight ROS environments for development and simulation. You are not tied to any particular version of Linux for the host and don't have to worry about "breaking" your environment. Word on the street is that you can now use Mac OSX and Windows as a container host yet run containers from Linux using what can only be described as straight up witchcraft.
2) Building an OpenGL enabled Docker container running ROS Indigo. This allows hardware accelerated 3D graphics (eg, Gazebo) under Docker.
3) Starting your container and attaching additional bash shells.
4) Mapping filesystems and devices in from the host, useful for keeping persistent data (like your Git / Hg tree or catkin workspace!) and connecting hardware devices to the container.
5) Cleaning up dangling images created by the build process.

I didn't really touch on it, but by default your Docker containers are running on a private subnet (rather than bridged to the host). Look into the "--net" parameter for docker-run for other options.