Please ask about problems and questions regarding this tutorial on answers.ros.org (http://answers.ros.org). Don't forget to include in your question the link to this page, the versions of your OS & ROS, and also add appropriate tags.

Using GUI's with Docker

Description: This tutorial walks you through using graphical user interfaces with Docker for various ROS tools.

Keywords: ROS, Docker, GUI, Tooling

Tutorial Level: INTERMEDIATE

In this tutorial, we go over some of the recent methods in enabling the use of graphical user interfaces within Docker containers. This is not perhaps not one of the intended use cases for Docker, but as Docker experimentation progressed, this has become a popular method to leverage and enable portable GUI applications. The methods listed are not exhaustive, as this all still quite new and continually evolving. Please feel free to contribute by keeping this wiki update and adding additional resources.

目录

Using X server

X server is a windowing system for bitmap displays, common on linux operating systems. There are several ways one can connect a container to a host's X server for display. A brief description and tradeoffs for each method below:

- The first listed is simple, but unsecure
- The second is safer, but non-isolated
- The third is isolated, but not as portable
- The fourth is isolated, works remotely, but is slow.

1. The simple way

The simple way is expose your xhost so that container can render to the correct display by reading and writing though the X11 unix socket.

```
docker run -it \
    --env="DISPLAY" \
    --env="QT_X11_NO_MITSHM=1" \
    --volume="/tmp/.X11-unix:/tmp/.X11-unix:rw" \
    osrf/ros:indigo-desktop-full \
    rqt
export containerId=$(docker ps -l -q)
```

Above, we made the container's processes interactive, forwarded our DISPLAY environment variable, mounted a volume for the X11 unix socket, and recorded the container's ID. This will fail at first and look something like this, but that's ok:

```
No protocol specified
rqt: cannot connect to X server unix:0
```

We can then adjust the permissions the X server host. This is not the safest way however, as you then compromise the access control to X server on your host. So with a little effort, someone could display something on your screen, capture user input, in addition to making it easier to exploit other vulnerabilities that might exist in X.

```
xhost +local:root # for the lazy and reckless
```

If you are concerned about this (as you should be), you have at least two options. The first is to run

```
xhost -local:root
```

after you are finished using the containerized GUI, this will return the access controls that were disabled with the previous command.

A better option is opening up xhost only to the specific system that you want, for instance if you are running a container on the local host's docker daemon with container's ID stored to the shell variable containerId

```
xhost +local:`docker inspect --format='{{ .Config.Hostname }}' $containerId`
docker start $containerId
```

This will add the container's hostname to the local family's list of permitted names.

2. The safer way

A another way is to use your own user's credentials to access the display server. This involves mounting additional directories:

```
docker run -it \
    --user=$USER \
    --env="DISPLAY" \
    --volume="/etc/group:/etc/group:ro" \
    --volume="/etc/passwd:/etc/passwd:ro" \
    --volume="/etc/shadow:/etc/shadow:ro" \
    --volume="/etc/sudoers.d:/etc/sudoers.d:ro" \
    --volume="/tmp/.X11-unix:/tmp/.X11-unix:rw" \
    osrf/ros:indigo-desktop-full \
    rqt
```

Some applications expect a home directory for the user in order to save and read configuration files, so if you attempt to use them without such a directory existing in the container's filesystem, you may receive warnings or errors.

You could chose to go one step further by mounting your own home directory into the container. This allows you access to local config file for your local user, maintaining the same username, password and file permissions.

```
docker run -it \
    --user=$USER \
    --env="DISPLAY" \
    --workdir="/home/$USER" \
    --volume="/home/$USER:/home/$USER" \
    --volume="/etc/group:/etc/group:ro" \
    --volume="/etc/passwd:/etc/passwd:ro" \
    --volume="/etc/shadow:/etc/shadow:ro" \
    --volume="/etc/sudoers.d:/etc/sudoers.d:ro" \
    --volume="/tmp/.X11-unix:/tmp/.X11-unix:rw" \
    osrf/ros:indigo-desktop-full \
    rqt
```

This last bit how ever removes quite a few layers of separation between what runs in the container and the environment of the host, and is thus not as isolated. So by means of convenience and security, one can lose some aspects of isolation, and other useful properties of repeatability, reducibility, and portability if not careful.

3. The isolated way

There is another way to emulate the same technique with the previous method but in a more isolated manner. We can do this with some modifications to the original image by creating a user with uid and gid matching that of the host user. This is an example of what you may need to add to the docker file, or simalurly run and commit in the container:

```
#Add new sudo user
ENV USERNAME myNewUserName
RUN useradd -m $USERNAME && \
        echo "$USERNAME:$USERNAME" | chpasswd && \
        usermod --shell /bin/bash $USERNAME && \
        usermod -aG sudo $USERNAME && \
        echo "$USERNAME ALL=(ALL) NOPASSWD:ALL" >> /etc/sudoers.d/$USERNAME && \
        chmod 0440 /etc/sudoers.d/$USERNAME && \
        # Replace 1000 with your user/group id
        usermod  --uid 1000 $USERNAME && \
        groupmod --gid 1000 $USERNAME
```

You may need to change the number 1000 for the uid and gid to mach that of your host's user.

The next step is to make a X authentication file with proper permissions and mount this to a volume for the container to use. Here is an example of a run command doing just this:

```
XSOCK=/tmp/.X11-unix
XAUTH=/tmp/.docker.xauth
touch $XAUTH
xauth nlist $DISPLAY | sed -e 's/^..../ffff/' | xauth -f $XAUTH nmerge -

docker run -it \
        --volume=$XSOCK:$XSOCK:rw \
        --volume=$XAUTH:$XAUTH:rw \
        --env="XAUTHORITY=${XAUTH}" \
        --env="DISPLAY" \
        -user="myNewUserName" \
    osrf/ros:indigo-desktop-full \
    rqt
```

Now the container is predominantly isolated with only read and write access to X authentication and socket file. The drawback of all this with is that some user specific configuration now resides the image itself, and thereby making it less portable. Should a different user, even on the same host machine, wish to use the same image, they will need to: start an interactive terminal session with the container, change the uid and gid to match their own, commit the container to a new image, and launch the desired GUI container from that one instead. Doing this back and forth also adds needless layers to your image, so introspecting the changins in an image would become a noisy affair.

There are clever ways to get around this, such as making a new user with the same uid and gid at runtime. This take a bit more entrypoint scripting and machinery, but can provide a more portable solution plus remaining just as isolated. An excellent example of this is docker-browser-box (https://github.com/sameersbn/docker-browser-box).

4. The ssh way

One of the first ways used to view GUI within containers was done using basic X11 forwarding using an ssh connection. This is bit more involved, as the number of moving parts increases.

We'll need a ssh installed and a running dedicated daemon within each container we launch. This also works against a bit of Docker zen in keeping one container limited to one process, as the daemon adds more dependencies and consumes additional resources.

We'll also need to specify what ports to map/expose for the container, and not to mention passwords or keys on both sides if we want to be secure about things. Its a bit fragile, as every time you spin a container from the same image, you may receive a different IP, an address you need to query Docker for, or look up from inside the container each time. Frame rates will be slow as the display is piped through the local network interface, and not a unix socket, rendering it unfavorable for high bandwidth imagery or user interfaces that deteriorate due to lag.

On the plus side, this does afford you the use of shell access and graphical interfaces within the container even when viewing from a remote client and/or different operating system. But perhaps such efforts could be also applied host's display itself (i.e. VNC), and resort to simpler methods above for connecting the containers to the host's display server. This would most likely depend on how you see your application being used, and what level of effort you'd be willing to put forth in using it.

A good example here has a detailed set instruction on how to enable X11 forwarding using Docker containers: Docker Desktop over SSH (http://blog.docker.com/2013/07/docker-desktop-your-desktop-over-ssh-running-inside-of-a-docker-container/)

Using VNC

Another common way of viewing graphical interfaces as well as virtual desktops with containers was done early on using Virtual Network Computing (VNC) protical. If low frame rates are not an issue, and you'd prefer a full desktop like view of a container's display, then using VNC may be a suitable option. Granted, it'll take much of the same upfront setup effort as using X11 forwarding with ssh, but you may gain better frame rates thanks to available color compressions and display downsampling settings.

You'll need to install a VNC server inside the image and make sure it starts and that the proper ports are exposed when you launch the container. At this point the container is basically running it's own X server, so it'll have quite a few additional processes running inside at this point. On the plus side, VNC clients are widely available and quite mature on multiple platforms, including phones, so connecting to VNC session is rather easy.

Here is a good example on how to enable VNC server in Docker containers: docker-ubuntu-vnc-desktop (https://github.com/fcwu/docker-ubuntu-vnc-desktop)

Using Wayland

Using Mir

Using Audio

1. Alsa Audio

2. Pulse Audio

Troubleshooting

A few topics to note

- The image you use should include all the graphical and display dependencies you require for runtime.
- Qt can be a bit buggy in rendering interfaces when used in this way. If you get an error like this:

```
X Error: BadAccess (attempt to access private resource denied) 10
  Extension:    130 (MIT-SHM)
  Minor opcode: 1 (X_ShmAttach)
  Resource id:  0x3800003
```

  You may need to include

```
 --env="QT_X11_NO_MITSHM=1"
```

  in your docker run parameters to set the environmental flag to force Qt to show properly.

Refs: http://olivier.barais.fr/blog/posts/2014.08.26/Eclipse_in_docker_container.html (http://olivier.barais.fr/blog/posts/2014.08.26/Eclipse_in_docker_container.html) https://github.com/sameersbn/docker-browser-box (https://github.com/sameersbn/docker-browser-box) https://github.com/jfrazelle/dockerfiles/tree/04454ac147edd99c40e42e0117d5c8e0539f5fca (https://github.com/jfrazelle/dockerfiles/tree/04454ac147edd99c40e42e0117d5c8e0539f5fca)

Wiki: docker/Tutorials/GUI (2017-10-05 18:58:13由jarvisschultz (/jarvisschultz)编辑)

Brought to you by:  Open Source Robotics Foundation

(http://www.osrfoundation.org)