14-848: Cloud Infrastructure (Fall 2019)
Project 3: Containers!

Important Dates

- Released: November 4, 2019 at 11:59 pm
- Due: November 21, 2019 at 11:59 pm

Overview

In class we discussed the operating system abstractions and constructs that support containerization and how they can be integrated to form a container abstraction. We also discussed the orchestration of containers in real world deployments.

This project is designed to solidify your understanding of the mechanics of each of these aspects of the implementation of containers and orchestration tools.

- In part 1, you'll implement a container facility from underlying operating system constructs
- In part 2, you'll develop a rudimentary orchestration system for the containers you created in part 1. This includes both an API and a set of command-line tools implemented using it.

Technical Logistics

Environment

Containers rely on OS-specific constructs. One consequence of this is that we do need to coerce everyone to work in exactly the same environment or things might not work when we test. Since we know that many people like working on individual hardware instead of the unix.andrew cycle servers, we could not select that environment as it uses a licensed distribution.

As a readily available alternative that should work for everyone, we selected Ubuntu 16.04 (Xenial Xerus). If that doesn't happen to be the operating system you use, that's okay. We don't expect that it is. We picked it because virtual machines are readily available for it, canned and ready to go.

The steps below are likely to work for most folks. But, you might need to adapt them for your environment. Or, you might prefer to go at it differently. Any way you get to having the right development environment is fine with us. But, please, for your own sake, don't try to start the project in a different environment and port it later. That path could be very painful.

Continued on next page.

Suggested steps to create the required VM:

- Download the 64 bit Desktop iso image:
  - http://releases.ubuntu.com/16.04/
- Read the following tutorial and follow the steps as described:
  - https://medium.com/@tushar0618/install-ubuntu-16-04-lts-on-virtual-box-desktop-version-30dc6 f1958d0
- *Hint*: If after booting up the VM, the display screen appears to be too small, try the following:
  - VM Settings >> Display >> Scale Factor = 200 %
  - VM Settings >> Display >> Screen >> Graphics Controller = VBoxVGA

3rd Party Libraries?

Unlike the BigTable project, since this project is not being tested on the linux timeshares, you can choose to use any third party library you require. See the section about the *Makefile* for more details. But do note that, in our experience with python3, we didn't require any third party library.

Part 1 – Implementing a Container Facility

Part 1 asks you to implement a container facility from the underlying operating system constructs. This part of the assignment asks you to read and follow the following blog/tutorial:

- https://ericchiang.github.io/post/containers-from-scratch/

Please note the following updates:

If the following command doesn't work for you:
sudo chroot rootfs python -m SimpleHTTPServer
Try the following instead:
sudo chroot rootfs /bin/bash -c "PYTHONHASHSEED=0 python -m http.server 8080"

If the following command doesn't work for you:
sudo mkdir readonlyfiles
Try the following instead:
mkdir readonlyfiles

Part 1 Deliverable

A file called "part1.txt" containing your command history as you complete the tutorial:

- This history need not contain commands entered before or after the tutorial.
- The history should not be edited other than to select only the time frame associated with completing the tutorial. Specifically, it should contain any mistakes, spurious commands, etc.

Continued on next page.

Helpful hint:

If you follow the instructions provided in the setup section of this writeup and use bash as your shell, history should be enabled, which will make it easy for you to collect the command history, e.g. “history > part1.txt”, and then edit out extra stuff at the beginning.

To be safe, please check to make sure history is enabled for your shell before starting. If you need help getting history turned on, please feel free to ask the course staff, or anyone else. The following forum threads below may be of interest to *bash* users.

- https://askubuntu.com/questions/261407/how-to-save-terminal-history-manually
- https://unix.stackexchange.com/questions/1288/preserve-bash-history-in-multiple- terminal-windows

Part 1 Reflection:

Following are a set of questions to think about as you complete and after you complete the tutorial. Note that answering these questions is not part of the deliverable. They are just intended to help you get the most out of part 1. Feel free to discuss these questions on piazza after you have made a fair attempt on your own.

(1) As mentioned in the updates above, the command “*sudo chroot rootfs python -m SimpleHTTPServer”* fails. The error it produces is *Failed to open: /dev/urandom*. Can you think of a reason why? Note that the directory “*/dev/urandom”* is present on the host system and you can “*ls -ld /dev/urandom”* to see the permission set on it.

(2) Why is the command “unshare” named “unshare” and not something like “cnn”, e.g. “create a new namespace”? More specifically, what pattern/model is used to create namespaces? Thinking about the fork-exec model might be of help to you.

(3) Think about the PID name space. What happens if you create a process, i.e. *vim*, from within a container? Does it show in *ps* from within the container? What about from the host outside of the container? Can you kill that instance of *vim* from another container? From the host container? From the container it within which it was created? Think about your answers – and then give it a try.

(4) Consider the mounting example in the tutorial, can you perform the mount after performing the chroot, i.e., from within the container? Why or why not? Think about your answer and then give it a try.

Continued on next page.

Part 2 - Container Management/Orchestration

This part of the assignment asks you to build a very primitive orchestration system and containerize an application using it. It is a very, very primitive version of a real-world system such as Kubernetes. As we discussed in class, real-world orchestration systems are critical management infrastructure for the cloud.

Specifically, you are asked to build a manager that create container configurations, launch container instances, list running instances, and kill running container instances.

The API

Much like P2, the primary interface to the management system will be via a standardized API. An API like this makes it possible to programmatically build up sophisticated and specialized management tools with UI's appropriate to the users.

In the context of this project, the API will be used to build up a set of command-line tools that can be used to control your management system, and also as a mechanism for it to integrate with our test set.

Rather than a language specific API, the project has a Web-based API. This is a very common idiom because it enables integration across many environments, programming languages, frameworks, and idioms. In the case of this assignment, it allows you to implement the project in any language you'd like – while ensuring that we can test it with a common set of test scripts. To achieve this, as is the case in the real world, the API has to be standardized – you can't change the specification (or our test scripts will not work).

- The API is described in the "API.md" file.

The CLI

This assignment asks you to implement a command-line interface (CLI) for the management. As is commonly the case in real world applications, such as Docker, the CLI should be implemented via the API. In other words, the commands should be implemented by calling API functions.

Please find below a list of commands that should be available in your CLI. Each provides access to exactly the same functionality as the corresponding API function. You are welcome to implement additional command-line commands as may be helpful to you. All commands exit status 0 upon success and non-zero upon failure.

The *launch* command is provided for you as an example. Look in the handout for "cli/launch".

Continued on next page.

Command-line interface Command List:

- upload [file]
  - Uploads a configuration file to the manager.
  - Hint: use the “config” REST API call
- cfginfo
  - Displays a list of configuration files, one per line
  - Files are named “name-major-minor.cfg”, e.g. “tiny-2-1.cfg”
- launch [name] [major] [minor]
  - Creates a new instance of a container
  - [name] is the name of the configuration
  - [major] is the major version number
  - [minor] is the minor version number
  - Upon failure, print “Failure: {some meaningful description}”
  - Upon success, print “Success: {InstanceName}”
  - Hint: a sample is provided in cli/launch
- list
  - Prints a list of running instances, one per line
  - The format should be, “InstanceName: {Associated configuration file name}”
- destroy [InstanceName]
  - Destroys the named instance
  - Upon success, print “Success”
  - Upon failure (including instance not found), print “Failure”
- destroyall
  - Destroys <u>all</u> running instances
  - Upon success (including nothing to destroy), print “Success”
  - Upon failure (including instance not found), print “Failure”

<u>Container configuration files</u>

A JSON-formatted configuration file specifies the configuration of a container.

- *name* – represents the name of the configuration
- *major* – represents the major version number of the configuration
- *minor* – represents the minor version number of the configuration
- *base_image* – specifies the base image to be read-only mounted. Analogous to the *rootfs* tar ball in part 1.
- *mounts* – a list of additional mounts that can specialize the base image: Analogous to mounting *readonlyfiles* in the tutorial in part 1, except these can be read or read-write as specified.
  - Read-only mounts can be shared, for bragging points. This is not a requirement.
  - Read-Write mounts are private, one per instance. They can be copy-on-write for bragging points, but this is not a requirement.
  - Mounts are performed in the same order listed.
- *startup_script* – the path to any start up script you’d like to run at launch. Optional.
- *startup_owner* – the userid of the user that should run the start up script. If this parameter is not provided, “root” should be assumed.
- *startup_env* – any environment variables you would like defined for your start-up script

Continued on next page.

Consider, for example, the configuration file below, which can also be found as “API.md” at the top level of the handout:

```
{
    "name": "sensiblename",
    "major": "1",
    "minor": "01",
    "base_image": "basefs.tar.gz",
    "mounts": [
        "webserver.tar /webserver/ READ",
        "homedir.tar /webserver/home READWRITE"
    ],
    "startup_script": "/webserver/tiny.sh",
    "startup_owner": "root",
    "startup_env": "PORT=8080;LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/var/lib64"
}
```

Manager Action On Instantiation

Upon instantiating a new container from a configuration file, the manager should do the following:

1. Create a directory where the container will be chroot’d and extracting basefs.tar to that container.
    - You can extract the *basefs* tarball to any directory on the container host you’d like. We suggest organizing them under a common directory.
    - If the mount point is not present, create it.
2. Apply each of the mounts with the read-only or read-write specification as requested.
3. Create a shell with the requested owner and environment and run the requested start-up script.

Hint:
If you are wondering how to run a shell command, such as *mount* or *tar*, from within the manager program, you are probably thinking in the right direction. Any programming language you’ll want to use has a mechanism for invoking shell commands programmatically.

For example, in the C Language, *system()* or (“man 3 system” for more information) or exec family (“man 3 execl” for more information) can be used. For another example, in Python, you can import the *os* module and then use the *os.system()* method to do the same. In the Go language, the *os/*exec package can be used.

Important: All of the mount files for step 2 above are provided to you with the handout. The base image required for step1 is not provided to you since it’s a large file > 200 MB. To download the base image, simple cd into the handout and type “make download”. You only need to do it once. Your manager should not re-download this file again and again over the network each time a container is being created. You can assume that make download will be run before you manager is started when testing your program.

Continued on next page.

Manager Configuration

In order to be accessible to clients, such as our test set, the manager need to be exposed over via a well-known hostname and port. To ensure that our test set can interact with your manager and CLI, please use the following values. They may be configuration constants, environment variables, put into a configuration file, or command-line arguments supplied by your Make file.

- MANAGER HOSTNAME = localhost
- MANAGER PORT = 8080

Also, remember that the manager needs to run with privilege. To achieve this, you'll probably want to use *sudo* in the Makefile that starts it. create containers.

Required container

Your manager should be able to create arbitrary containers, as described by the configuration files. Many of our tests rely upon the following configuration, so you'll want to test it specifically. It can also be found as "API.md" at the top level of the handout:

```
{
    "name": "sensiblename",
    "major": "1",
    "minor": "01",
    "base_image": "basefs.tar.gz",
    "mounts": [
        "webserver.tar /webserver/ READ",
        "homedir.tar /webserver/home READWRITE"
    ],
    "startup_script": "/webserver/tiny.sh",
    "startup_owner": "root",
    "startup_env": "PORT=8080;LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/var/lib64"
}
```

Please note:

- With the exception of *basefs.tar.gz*, we provided the associated tarballs in the handout.zip
- The base image required for step1 is not provided to you since it's a large file > 200 MB. To download the base image, simple cd into the handout and type "make download". You only need to do it once.
- Since the base image is read-only, your manager should not re-download this file over the network each time a container is being created. This is emphasized both to help you while developing – and us while testing.

Continued on next page.

Testing

Automated Testing

Our primary test scripts use the REST API:

- The style is very much the same as was the case for the BigTable project.
- The test scripts are written in Python3, which is provided by the recommended image.
- Our API-based testing scripts (Config test, launch test, and container test, isolation test) can be run as “make api_tests
- To aid with testing, we have placed some scripts in the “cgi-bin” folder of the *tiny* web server.
- In addition to the API-based automated testing, we have automated tests of the CLI. These can be run as “make cli_tests”. Remember that the CLI is implemented via the API.

Our test scripts have a similar pattern:

1. Create a configuration file and upload it to the manager.
2. Ask the manager to launch a web server container by specifying the same name, major and minor version from step 1.
3. Repeat 1 and 2 for multiple containers
4. Run tests to determine that the instances were created correctly and with correct isolation and that the specified start-up script ran correctly with the correct environment and user.
    - Our test scripts often communicate with the container via CGI scripts running under the containerized Web server. This enables them to, for example, enumerate processes in each container.

Hint: You probably want to read the testing and “cgi-bin” scripts to get an idea about how we are testing. They are not intended to be “black box”.

Rubric

Part 1	Bash History	15
Part 2	*Makefile* adjustment	10
	Config Tests	15
	Launch Tests	15
	Container Tests	15
	CGI Tests	15
	CLI tests	15
Total		**100**

Continued on next page.

Submission

There is no checkpoint for this project. As always, zip up all your files and submit on Canvas. Your zip should contain:

- The bash history for part 1 in a file named “part1.txt”.
- Source files for your manager and CLI
- The updated *Makefile*.
- An optional Readme.txt file in case you need to tell us something.

Note: Please do not include the base image in your submission. When we test your program, we will run make download to download basefs.tar.gz.

Have a fun time containerizing, :-)

“Appendix A: The tiny Web Server” on next page.

Appendix A: The Tiny Web Server

The container that we are using for most of our tests containerizes the *tiny* web server from 213/513/600/613.

As the name suggests, *tiny* is a small and simple web server capable of serving both static and dynamic content. For the purpose of this project, you don't need to understand it's implementation. You will need to know how to use it.

To run the tiny web server at a particular port (say 8080) use the command: ./tiny 8080. That's it. It's that simple. To check whether the server is running or not fire up a web browser and visit http://localhost:8080/.

Serving static content: If the request url is something like http://localhost:8080/var/index.html, the tiny web server will simply read the file /var/index.html and return it.

Serving dynamic content: If the request url contains "cgi-bin" following the root directory (/), the tiny web server will execute the specified executable from the cgi-bin and return it's output. For example, if the request url is something like http://localhost:8080/cgi-bin/myscript.sh, the tiny web server will execute the script myscript.sh and send over the network the output generated by that script.

"Appendix B: Roadmap" on next page.

Appendix B: Roadmap

A sample roadmap to part B of this project is as follows:

(0) Get the virtual machine up and running. Inside the virtual machine follow these instructions. Make sure that you thoroughly understand the steps you followed in Part 1 of the assignment.

(1) Download and extract the handout. In the extracted folder run **make download** to download the base image.

(2) Extract the base image and go through it's contents to get a feel for how the underlying file hierarchy looks like. In particular you will want to ensure that the executable webserver/tiny runs without containerization as well. Read the files webserver/cgi-bin/ps.sh, webserver/cgi-bin/pkill.sh and webserver/tiny.sh. Note that the script tiny.sh will not run unless you containerize but reading it will give you a sense of what the containerized environment should look like.

(3) Writing code for the manager:

- Start with reading API.md to understand what the expected REST interface is.
- Setup hooks (GET, DELETE, PUT, …) as required by API.md. Remember you run the manager at a fixed ip and port number as opposed to accepting them as arguments.
- Read the test cases and carry out the development following the same progression.

Test Case	**Interfaces tested**
ConfigTests	/config /cfginfo
LaunchTests	/launch /list /destroy /destroyall
ContainerTests	/launch + checks for whether the webserver is running or not
CGITests	checks for whether the web server was properly containerized or not

- Go through the section about manual testing to understand what the CGI tests do and what is the expected outcome.

(4) Writing the command line interface (CLI):

- Once you have the REST APIs working properly, it shouldn't be much work to create the command line interface.
- The CLI for the purpose of this project is simply a set of scripts (or executables) placed under the folder **cli**. The CLI tests (make cli_tests) invoke these scripts (or executables) written by you. For a sample, we have provided an implementation for the **launch** script.
- For each command line option there should be one script (or executable). So when you are done with this part, your cli folder should have scripts (executables) named **upload, launch**, **list**, **cfginfo**, **destroy**, **destroyall**.

- Going through the launch script, you will find it looks awfully a lot like the api_tests that we wrote. They are similar because they use the same set of REST APIs to talk to the manager. That's all there is to it. The command line scripts (or executables) are just a wrapper around the REST APIs.