Ben Rivera & Eddie Goode
Professor Dubey
CS 3281
4 May 2017

Linux Orchestration Framework Report

Introduction:

This is a report on our final project in our Operating Systems class: CS 3281. We were tasked with designing and creating an orchestration framework in Linux for running processes on multiple clients. We have created two C++ executable programs, one to run on the master node and one to run on a client node. The master node is responsible for reading in user input and delegating tasks to clients. This report includes an outline of our project's design, instructions on launching and shutting down processes, and solutions to several challenges we encountered in the creation of our project.

Project Design:

We have designed an orchestration framework, capable of launching processes on remote nodes and capturing log data. Multiple clients can connect to the master node, and the user can easily issue commands to any number of nodes. Nodes communicate with each other using TCP sockets. The master node and client nodes both establish TCP connections and run indefinitely until the user enters "exit." A list of available nodes is constantly maintained by the server and is updated frequently using "heartbeat" messages. To see a list of possible commands we have implemented, enter "h" in the bash prompt.

The client and master sides each have infinite loops for sending and receiving commands. In each of their start commands they run and parse the buffer/strings for each command and execute appropriately for each command. These designs can be further explored in our design document.

User Interaction Design:

When the master server class is started, a shell prompt is displayed to the user. Once clients connect to the master node, the user can is prompted to enter commands. If a user would like to add more nodes, the user must type exit. There are a set of coded keyword commands that execute functions we have built. To see a list of possible commands, enter "h". The commands we support include "copy" to copy a file to a remote node, "list" to return a list of the client nodes, "pid" to return a list of current processes and their statuses, "kill" to shut down a running process gently, sKill to force a shutdown, execute, remove, and "exit" to shutdown the master node. If the user enters a standard function like "ls" or "mkdir" then the command is executed on the remote node using the app() class which checks a list of builtin commands. We had ours designed this way because it allowed for better testing and functionality.

The format of commands can be conveniently found in our initial readme.

How to Launch a Process:

We have abstracted away much of the work in a launching a new process. The client program includes the app() class which is responsible for creating and executing processes. If the user enters a command to launch a new process on a client node, we pass the command to the execute() method of the app() class. The execute() method runs the command in background without. Before we call execvp we properly use fork and dup 2 to transfer our output to a logfile.

Testing:

To test our project, we utilized ssh to securely connect to the master node. Along with running our code in the terminal, we used the debugger available in cLion to test and fix problems we encountered. We tested each command by opening multiple clients and running the command. We then checked to see if they worked by looking in their respective directories and logfiles. Testing for this project was tedious, as you had to compile, copy cShell to respective locations, and then launch multiple nodes. We used the capabilities of cLion extensively. The debugger allowed us to debug the client side and the master side simultaneously.

Resource Usage and Process State:

The user can issue the command “getrusage” to return a list of current processes and their resource usage. Getrusage(2) is a Unix command that returns resource usages measures for running processes. By using the command with the RUSAGE_CHILDREN parameter, we are able to get the information of all of the running child processes running on a node. When the user issues this command, we run getrusage on every available node and return the information to the user. The second parameter of Getrusage is a struct that returns the respective information. We then manipulate that information and return it to the master node using our socket.

Process state queries were accomplished through using waitpid(PID, &status, WNOHANG) after execute and for each query. Our function getChildStatus(pid) was called. After an execution, it was only called on the pid of the executed process, and it did the proper signal_handling to get the status of the process. This was then manipulated into a format that was PID followed by status. That was then sent back to the mater. The master then added it to a map that was specific to that client. This map was in the clients struct and mapped each PID to its status. When a query was called and the user wanted to see each client and its current status, the server sent a request to the client that would cause the client to update the status through the method described above.

Shutting Down Processes:

The user has the ability to specify a running process and shut it down gracefully or forcefully. Unix includes POSIX signals to support asynchronous process communication. To shutdown a running process, the user can enter “kill <pid>” or “sKill <pid>” to shut it down. When the command is entered on the server side, we send the command to the node where it is then parsed. The client side then searches uses the linux system call kill(PID,SIGNAL). The only difference beteween the gentle and aggressive is their signal calls. Gentle = SIGINT, Forced = SIGKILL. In the event of an error, the error is logged to the log file and a response is sent back to the server.

Capturing Process Output and Dup2:

The app() class that is responsible for executing processes frequently issues output statements and error messages. These output statement are normally sent to stdout. To capture the output from the client, we create a file called “log.txt.” When a new process is created, we use dup2(1,”log.txt”) and dup2(2,”log.txt”) to send stdout and stderr to the file.

Future Implementations:

Given more time, we would like to improve the user interface of our project. Other orchestration frameworks like Apache Brooklyn have impressive GUIs for tracking nodes and easily issuing commands. Security wise we would like to ensure the user side can only accept a number of commands. Currently it can except more than exec and remove. Additionally, we would like to ensure that the socket’s port is private and that there can be no other node sending to the client node.