Project Final Report

CS 370 Fall 2021 Term Project - Audi Group

Due: 12.2.2021

Eric Martin

Blake Davis

Victor Berggren

Deploying Containerized FTP and NGINX™ Servers Across a Raspberry Pi Cluster Kubernetes Framework with Built in Physical Tamper Detection Sensing

Abstract

Utilizing Kubernetes as a framework to manage and cluster together multiple Raspberry Pi microcontrollers we have created a mini 'supercomputer' that can run load balanced containerized deployments simultaneously across all nodes in the cluster. The cluster runs both a file transfer protocol (FTP) server as well as an NGINX web server and can readily run other containerized Docker applications if needed. The device is also running a Python script that monitors for physical tampering of the device via an accelerometer. Since we are not running an SFTP server, the security was bolstered by the use of usernames, passwords, two-factor authentication, limiting login attempts, setting client-alive intervals, preventing root login, and equipping an accelerometer to detect physical tampering, and a display screen that shows all users who are currently logged in through SSH. The Informational LCD screen also displays the master node CPU load, IP address, disk space, memory usage, and CPU Temp. On-board sensors for the detection of physical tampering are found within the sense-hat shield accessory. The accelerometer found within the shield is utilized to detect any movement in the x, y, or z axis. Once movement is detected, the monitoring python script will alert all authorized via email that the cluster has been tampered with an associated system log.

Introduction

In order to utilize the computing power of multiple Raspberry Pi microcomputers one can create a cluster managed by the Kubernetes framework. Once clustered, the user can harness the combined computational power, cores, ram, and memory of the cluster to run parallelizable applications. This is not as simple as adding all of the resources of each device but the effect is similar. Alongside the combined computational power, another benefit of clustering devices is maximizing uptime and allowing for scalability. Applications that are designed so that they can scale up and down by parallelization of tasks are perfect for running on a cluster. Take an NGINX web server for instance. If there is only one web server, it could only be scaled up until you reach your RAM cap or require a faster CPU. But if multiple instances of the web server were running across a cluster, multiple requests could be handled by individual worker nodes and each node could take as much or as little resources as needed in aggregate. Until that is, a worker node starts slowing down for some reason, freezes, or randomly crashes or dies. In the case of the single web server, that would mean downtime for the server until the issue is resolved. But in the case of a cluster, the node can be easily fixed or replaced while the other nodes take on the work of the offline node. Managing the nodes in this scenario is easily handled with the Kubernetes platform. Since we are running on Raspberry Pi's, a lightweight Kubernetes distribution option is needed. we utilize K3s which is a highly available, certified Kubernetes distribution designed for production workloads in resource constrained IoT appliances. With K3s we can easily view and manage our containerized applications across the cluster. Utilizing containerized applications is a benefit because we can run identical Docker images across all nodes and not have to worry about the specific environment or installed dependencies of each node in order to run the application. In this report we will cover the process of setting up NGINX and FTP web servers across a Raspberry Pi cluster running on the Kubernetes platform and how to enhance the security of the cluster with both software and hardware changes.

Problem Characterization

The problem we set out to solve is two-fold. The first problem is that a single Raspberry Pi is a relatively weak computer on its own and most will struggle to run more demanding apps without overheating. This is due to the lack of resources that a full-sized desktop computer would have access to. Our solution to this issue is by simply adding more resources, which in this case is more Raspberry Pi's. By clustering three or more Raspberry Pi's into one system while taking advantage of Kubernetes ability to easily manage container orchestration we can easily handle app deployments that would not be possible to run on a single unit.

The second problem is that relying on third-party cloud servers for simple web hosting and file storage, while convenient, has its drawbacks. If you want to store private files, you must trust the third-party to handle your data responsibly and securely. These services also require you to pay a fee if you want to specify how your server will be set up, and even then, your customization is limited. A relatively cheap alternative is to create your own web hosting and FTP server. Your data will be stored on a machine you have complete control over and physical access to. There are countless measures you can take to make your server private, secure, and reliable, at the cost of some effort and time to research. This last point is more often the major drawback of self-hosting, as securing your servers is not a trivial process and security is an ever-evolving landscape. But for the few who enjoy the challenge, it is a highly rewarding process with great learning potential.

In order to manufacture such a device as defined in the above sections, the following steps were defined in the product design process:

1. Cluster three (or more) Raspberry Pi's using Kubernetes (K3s) to pool their resources
2. Implement an on-board sensor to alert managers of physical tampering
3. Set up an FTP server on the cluster that can be remotely accessed
4. Deploy a containerized, load-balanced NGINX web server across the cluster
5. Evaluate the performance of the cluster computer and its limitations'
6. Take steps to help secure unwarranted access to the cluster and its servers
7. Evaluate potential security flaws of the FTP server and cluster
8. Implement an on-board LCD screen to display system status information
9. Evaluate the cost and marketability of the entire setup

Out of these nine design steps, we were eventually able to accomplish all of them within a reasonable time frame. This was not without a vast majority of that time being spent troubleshooting though. We did not realize at first that in order to cluster Raspberry Pi devices with Kubernetes, the Raspberry Pi boards used must have a minimum of 4gb RAM each. Initially we only had one 2gb model 4 Pi and two 1gb model 3 Pi models to work with. Attempting to cluster these using K3s was unsuccessful, and we purchased two 4gb and one 8gb model 4 Raspberry Pi's to use in the cluster. The cost of these boards will be disclosed in the cost and marketability section. The execution of the design steps will be discussed in the following sections.

Materials, Costs, and Resources

The list of materials and their costs utilized in the project are listed in the table below. The table below details the specific items used in this project. Many of the items were already owned by the team. The items that were purchased specifically for this project were the Raspberry Pi boards, totaling $338. The use of individual Raspberry Pi cases (which is presented in this project) is not advisable as they do not easily stack and take up more space than a Raspberry Pi server case which is designed to hold multiple boards in a streamlined fashion. The various resources cited in this project can be found in the references section.

Amount	Approx. Cost	Item Description
1	$118 (for 1)	Raspberry Pi 4 Model B 8gb RAM with power and HDMI cables
2	$220 (for 2)	Raspberry Pi 4 Model B 4gb RAM Cannakit with case and cables
3	$30 (for 3)	Micro SD cards with 32gb memory
3	$15 (for 3)	USB Memory Sticks with 16gb Memory
1	$80 (for 1)	2TB External Hard-Disk-Drive with External Power Supply
1	$40 (for 1)	Raspberry Pi Metal Case with RGB Copper-Cooling Fan
1	$30 (for 1)	Raspberry Pi Sense-Hat Shield
1	$15 (for 1)	LCD Screen - Adafruit ILI9341 3.2” TFT Display with Touchscreen
TOTAL	$548	Cost of essential items only for 3 Pi Cluster: ~ $300 - $375

Solution and Implementation Strategy

Steps for deploying Containerized FTP and NGINX™ Servers Across a Raspberry Pi Cluster managed by a K3s based Kubernetes Framework with Built in Physical Tamper Detection Sensing and LCD display are as follows:

1. Installing a headless version of Raspbian OS image on each Pi
2. Configuring each Pi to allow hierarchical management and allocation of system resources and set kernel to 64bit
3. Connecting each Pi to Wi-Fi and assigning them static IP addresses
4. Prepare each Pi for K3s installation
5. Installing K3s and Forming a Kubernetes cluster in a master-worker relationship
6. Deploying an NGINX server across our Kubernetes cluster
7. Installing an FTP server on the master node that is interfaced with through FileZilla on our local machines
8. Implementing an accelerometer to detect physical tampering of the cluster
9. Developing a Python program that interacts with the accelerometer to send an email with an authorization log informing us that the cluster has been tampered with
10. Set up an LCD screen that displays system information
11. Secure the cluster

Methodology:

1) Installing a headless version of Raspbian OS image on each Pi

1) Login
    a. `Username: pi`
    b. `Password: raspberry`
2) Set new password
    a. `$ passwd`
3) Set configurations
    a. `$ sudo raspi-config`
        i. `System Options > Wireless LAN`
        ii. `System Options > Hostname`
        iii. `Interface Options > Enable SSH`
        iv. `Localization Options > Time zone`
        v. If keyboard is inputting the '@' sign wrong or the '|' sign
            1. `Localization Options > Keyboard > Enter > US > Defaults`
4) Update / Upgrade
    a. `$ sudo apt update`
    b. `$ sudo apt upgrade`

2) Configuring each Pi to allow hierarchical management and allocation of system resources and set kernel to 64bit

1) Open cmdline.txt from /boot/ and add the following to the end
    a. `cgroup_memory=1 cgroup_enable=memory`
2) open config.txt from /boot/ and add the following at the bottom
    a. `arm_64bit=1`
3) Reboot
    a. `$ sudo reboot`

3) Connecting each Pi to Wi-Fi and assigning them static IP addresses

1) Raspberry Pi's current IP address
    a. `$ hostname -I`
2) Router's gateway IP address
    a. `$ ip r | grep default`
3) Current DNS IP address
    a. `$ sudo nano /etc/resolv.conf`
4) Add Static IP Settings
    a. `$ sudo nano /etc/dhcpcd.conf`
        i. `interface wlan0`
        ii. `static ip_address=<your IP address>/24`
        iii. `static routers=<Routers gateway IP>`
        iv. `static domain_name_servers=<DNS IP address>`
5) Open cmdline.txt again and add the following to the end
    a. `ip=<your_static_IP>::<Gateway_IP>:255.255.255.0:<hostname>:wlan0:off`
        i. hostname is same as the one you setup in raspi-config
6) Reboot
    a. `$ sudo reboot`

4) Prepare each Pi for K3s installation

1) Prep for K3s Kubernetes Distribution
    a. Login or SSH into your Pi
    b. Configure legacy IP tables (if running earlier Raspbian release than Bullseye)
        i. `$ sudo iptables -F`
        ii. `$ sudo update-alternatives --set iptables /usr/sbin/iptables-legacy`
        iii. `$ sudo update-alternatives --set ip6tables /usr/sbin/ip6tables-legacy`
        iv. `$ sudo reboot`

5) Installing K3s and Forming a Kubernetes cluster in a master-worker relationship

1) Install K3s and become **Master Node**
    a. Become root
        i. `$ sudo su -`
    b. Install K3s master setup
        i. `$ curl -sfL https://get.k3s.io | K3S_KUBECONFIG_MODE="644" sh -s -`
            1. *the <`K3S_KUBECONFIG_MODE="644"`> part lets us use Rancher*
    c. Save the master node token
        i. `$ sudo cat /var/lib/rancher/k3s/server/node-token`
2) Install K3s and become **Worker Node**
    a. `$ curl -sfL https://get.k3s.io | K3S_TOKEN="YOURMASTERTOKEN" K3S_URL="https://[your server]:6443" K3S_NODE_NAME="servername" sh -`

    For example:

    b. `$ curl -sfL https://get.k3s.io | K3S_TOKEN="K10d2ea09c56bd44cbefe1721206bbb2bdabeeb6e1b4167dfacdf0286e7989fb03a::server:3a35baf18fd250b87ff73d7364afa348" K3S_URL="https://10.255.234.91:6443" K3S_NODE_NAME="harry" sh -`

6) Deploying an NGINX server across our Kubernetes cluster

1) Deploy first NGINX server app across the cluster
    a. Copy and paste the `nginx_server.yaml` into the root directory of the Master Node
        i. See attached file nginx_server.yaml
    b. Deploy app
        i. `$ kubectl apply -f nginx_server.yaml`
    c. Check status of application pods
        i. `$ kubectl get pods`
2) Expose NGINX containerized apps with a Node Port so it is accessible
    a. Copy and paste the `nginx_nodeport.yaml` into the root directory of the Master Node
        i. See attached file nginx_nodeport.yaml
    b. Deploy YAML
        i. `$ kubectl apply -f pensieve.yaml`
    c. Check status of deployment
        i. `$ kubectl get services`
3) Test the if you can access the NGINX server through localhost
    a. Visit the IP address of any Node through port 31111
        i. Example `10.255.234.222:31111`
4) To terminate a running app if needed:
    a. `$ kubectl delete -f nginx_nodeport.yaml`

7) Installing an FTP server on the master node that is interfaced with through FileZilla on our local machines

1) Install FTP Server Utility on Master Node
   a. `$ sudo apt install vsftpd`
2) Edit the configuration file
   a. `$ sudo nano /etc/vsftpd.conf`

   The following settings lock the server users to the FTP folder within the home directory.

   Uncomment the following:

   b. `# write_enable=YES`
   c. `# local_umask=022`
   d. `# chroot_local_user=YES`

   Change the following:

   e. `anonymous_enable=YES` to `anonymous_enable=NO`

   Add the following lines to the end of the configuration file:

   f. `user_sub_token=$USER`
   g. `local_root=/home/$USER/FTP`

3) Create FTP Directory
   a. `$ mkdir -p /home/[user]/FTP/[subdirectory_name]`
4) Modify directory Permissions (remove write permission)
   a. `$ chmod a-w /home/[user]/FTP`
5) Restart Vsftpd Daemon
   a. `$ sudo service vsftpd restart`
6) Install FileZilla on a remote machine
   a. `$ sudo apt install filezilla`
7) Establish Connection
   a. Open FileZilla on remote machine
   b. Enter Master Node IP address, Username, Password, and Port Number
   c. Click Quickconnect
      i. Transfer files to cluster

8) Implementing an accelerometer to detect physical tampering of the cluster

1) Install SenseHat onto selected node
   a. `$ sudo apt install sense-hat`
   b. https://pythonhosted.org/sense-hat/api/#imu-sensor

9) Developing a Python program that interacts with the accelerometer to send an email with an authorization log informing us that the cluster has been tampered with

a) See attached files
   a. Hogwarts_Security_Protocol.py
   b. *Send_email.py*

10) Set up an LCD screen that displays system information

a) See attached files

   a. lcd_sys_info.py

11) Secure the cluster

1) In `etc/ssh/sshd_config`
   a. Disable root ssh login
      i. `PasswordAuthentication no`
   b. Only allow 10 login attempts
      i. `MaxAuthTries 10`
   c. Poll clients every hour for activity before logging them out
      i. `ClientAliveInterval 1800`
2) Enable 2FA using google authenticator
   a. `$ sudo apt install libpam-google-authenitcator`
   b. execute the following to create token data:
      i. `$ google-authenticator`
   c. Save your secret key and emergency scratch codes
   d. Answer the following questions:

```
1)  Do you want authentication tokens to be time-based (y/n) y
```

```
2)  Do you want me to update your "/home/user/.google_authenticator" file (y/n) y
```

```
3)  Do you want to disallow multiple uses of the same authentication token? This restricts you
    to one login about every 30s, but it increases your chances to notice or even prevent man-
    in-the-middle attacks (y/n) y
```

```
4)  Do you want to disallow multiple uses of the same authentication token? This restricts you
    to one login about every 30s, but it increases your chances to notice or even prevent man-
    in-the-middle attacks (y/n) y
```

```
5)  By default, tokens are good for 30 seconds. In order to compensate for possible time-skew
    between the client and the server, we allow an extra token before and after the current
    time. If you experience problems with poor time synchronization, you can increase the
    window from its default size of +-1min (window size of 3) to about +-4min (window size of
    17 acceptable tokens).Do you want to do so? (y/n) y
```

```
6)  If the computer that you are logging into isn't hardened against brute-force login
    attempts, you can enable rate-limiting for the authentication module. By default, this
    limits attackers to no more than 3 login attempts every 30s.Do you want to enable rate-
    limiting (y/n) y
```

3) Make a backup of PAM's SSH configuration file `/etc/pam.d/sshd`:
    a. `$ sudo cp --archive /etc/pam.d/sshd /etc/pam.d/sshd-COPY-$(date +"%Y%m%d%H%M%S")`
4) Now we need to enable it as an authentication method for SSH updating `/etc/pam.d/sshd`:
    a. `$ sudo vim /etc/pam.d/sshd`

    now change

```
# Standard Un*x authentication.
@include common-auth
```

to

```
# Standard Un*x authentication.
#@include common-auth
auth required pam_google_authenticator.so
```

5) Tell SSH to leverage it by editing `/etc/ssh/sshd_config`:
    a. `$ sudo vim /etc/ssh/sshd_config`

    Comment out the last three lines:

```
#ChallengeResponseAuthentication yes
#UsePAM yes
#AuthenticationMethods publickey,keyboard-interactive
```

to

```
ChallengeResponseAuthentication yes
UsePAM yes
AuthenticationMethods publickey,keyboard-interactive
```

6) Restart ssh:
    a. `$ sudo service sshd restart`

ii. Libraries Used

- **send_email.py**
    - Smtplib
        - The smtplib module defines an SMTP client session object that can be used to send mail to any internet machine with an SMTP or ESMTP listener daemon.
    - Pathlib
        - This module offers classes representing filesystem paths with semantics appropriate for different operating systems. Path classes are divided between pure paths, which provide purely computational operations without I/O, and concrete paths, which inherit from pure paths but also provide I/O operations.
    - email.mime.base
        - This is the base class for all the MIME-specific subclasses of Message. Ordinarily you won't create instances specifically of MIMEBase,

        - although you could. MIMEBase is provided primarily as a convenient base class for more specific MIME-aware subclasses.
    - email.mime.multipart
        - A subclass of MIMEBase, this is an intermediate base class for MIME messages that are not multipart. The primary purpose of this class is to prevent the use of the attach() method, which only makes sense for multipart messages. If attach() is called, a MultipartConversionError exception is raised.
    - email.mime.text
        - A subclass of MIMENonMultipart, the MIMEText class is used to create MIME objects of major type text. _text is the string for the payload.
    - email
        - The email package is a library for managing email messages. It is specifically not designed to do any sending of email messages to SMTP (RFC 2821), NNTP, or other servers; those are functions of modules such as smtplib and nntplib.
    - Subprocess
        - The subprocess module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes.
    - time
        - This module provides various time-related functions.

- **Hogwarts_Security_Protocol.py (sensor script)**
    - sense_hat
        - This library is used to facilitate the communication between the Raspberry Pi and the Sense-Hat shield
    - digitialio
        - The digitalio module contains classes to provide access to basic digital IO on the Raspberry Pi.
    - Board
        - The board module is built into CircuitPython and is used to provide access to a series of board-specific objects, including pins.
    - time
        - see above definition
    - signal
        - The signal library allows defining custom handlers to be executed when a signal is received.
- **lcd_sys_info.py (LCD Screen script)**
    - PIL

        - PIL is the Python Imaging Library which is used for the LCD
    - adafruit_rgb_display.ili9341
        - This library is used to facilitate the communication between the Raspberry Pi and the Adafruit ili9341 LCD screen
    - board
        - see above definition
    - digitialio
        - see above definition
    - subprocess
        - see above definition
    - time
        - see above definition
    -

iii. Code

- **send_email.py**
- **Hogwarts_Security_Protocol.py**
    - **auth_logs.sh**
- **lcd_sys_info.py**
- **nginx_server.yaml**
- **nginx_nodeport.yaml**

iv. Presentation

- **PowerPoint Presentation of Device**

Evaluations

This project has a lot of interesting components and potential for evaluation:

i. Performance

We evaluated the limitations of our supercomputer by stress testing it and comparing its performance to a non-clustered Raspberry Pi. We were curious how the addition of 2 Raspberry Pi helper nodes and the container orchestration of Kubernetes (K3s) would improve performance compared to running the same apps on a standard Pi running Raspbian OS. For our setup, running NGINX and the FTP server, the cluster and the stand-alone pi ran at roughly the same speed because both of the processes that were running were relatively small and did not need much “horsepower” to run to completion.

ii. File Transfer Speed

We evaluated the limitations of the FTP server on our master node. We sent our server various file types and sizes to study and evaluate the performance of our FTP server. We hypothesized that the FTP server we built would perform slower than our desktop computers. Our hypothesis was correct, it took a bit longer for the FTP server to receive and download the file to it, compared to sending it to another one of our desktop computers. Sending it to the desktop computer is still slower than email because we have to account for the time it takes to upload it to the server over the internet.

iii. Security

We evaluated the potential security holes in our cluster computer, from both a physical and software perspective. Given the small form-factor of the device, its physical security is dependent mostly on the security of the place it's stored in and whether unauthorized access to the device can be prevented. If physical security is a concern, preventative measures can be taken such as storing it in a locked room. We have integrated a reactive security measure by sending out an email alert when the device is physically moved. This is done with two Python scripts and a sense-hat accelerometer and allows for instant response to any unauthorized physical access.

On the software side, we restricted the ability to SSH into the system to specific user groups, and we disable the ability to login as root via SSH. Additionally, we implemented two-factor authentication, and configured it to disallow multiple users using the same auth token, and rate-limited the auth module to only allow 3 login attempts per 30 seconds. These two measures help combat man-in-the-middle attacks and brute-force attacks against our server and adding 2FA means even if our credentials were stolen our server would not be immediately compromised.

There are still security vulnerabilities that are yet to be addressed, however. First off, all users currently have *sudo* permissions and have root access. This may or may not be a big security issue depending on how many users have been created and who has access to the server, but it could be wise to remove *sudo* permissions from certain users or restrict it to a single user. The major security vulnerability is that our file server is using FTP (File Transfer Protocol). FTP does not use encryption and sends login credentials in plaintext. This protocol is outdated and our first step in further improving our security should be to upgrade to SFTP, which is an extension of the SSH protocol. Other viable protocols exist as well, such as SCP and FTPS.

iv. Usability and Marketability

We evaluated the cost and marketability of a device based on our project. We envision a product that is small, compact, and portable that is an immediate file server anywhere you take it, as long as you have a stable internet connection or a portable hotspot. You could also dock this file server at home and SSH into it from anywhere in the world. Because it is small and compact, storing it in your house is simple as it could just sit on a shelf, much better than having a large server room taking up precious space in your house. If you move or choose to give it to your friend, it is as simple as unplugging a few cords and taking it to its new destination.

This product would be marketable to people that would prefer to have a home mini-server rather than using a cloud service such as Google Drive or Dropbox. The device can even be used to host video-game servers. We like the security that comes from storing your server in your own home and being able to secure it both physically and digitally. While this product would require minimal set-up from the user, it would be fully customizable as if the customer had installed all the libraries and followed our exact methodology themselves. This allows each customer complete control over their server configuration. Aside from being used as a server, the device setup lends itself to be utilized as a mini ‘supercomputer’ and can be used to run parallelizable apps, set up multiple sensors, or even mine cryptocurrency.

Conclusions

The resulting Kubernetes cluster successfully acts as an FTP that is accessible from outside of the local area network. The cluster also hosts an NGINX web server that is running in across all nodes with active load-balancing. The cluster stack also runs a physical tampering accelerometer sensor with the capability to email authorized users when activated. The cluster also hosts an LCD screen that displays the system vitals of the Master node and who is currently connected to the Node via SSH. The cluster’s security has also been hardened and now requires two-factor-authentication in order to access via SSH.

References

K3's Project Authors. (n.d.). K3s: Lightweight Kubernetes. K3's Documentation. Retrieved November 20, 2021, from https://k3s.io/

Marijan, B. (2021, April 6). How to Set up FTP Server on Your Raspberry Pi. Knowledge Base by PhoenixNAP. Retrieved November 20, 2021, from https://phoenixnap.com/kb/raspberry-pi-ftp-server

Network Chuck - Learn. (n.d.). Network Chuck. Retrieved November 20, 2021, from https://learn.networkchuck.com

Raspberry Pi. (n.d.-a). Build an OctaPi. Raspberry Pi Documentation. Retrieved November 20, 2021, from https://projects.raspberrypi.org/en/projects/build-an-octapi/0

Raspberry Pi. (n.d.-b). Raspberry Pi Documentation - Remote Access. Raspberry Pi Documentation. Retrieved November 20, 2021, from https://www.raspberrypi.com/documentation/computers/remote-access.html

PythonHosted.org - Raspberry-Pi Sense-Hat API Reference https://pythonhosted.org/sense-hat/api/#sense-hat-api-reference

Enzo Barrett – Secure Your Own Linux Server – HashDump – Colorado State University https://www.cs.colostate.edu/~enzob/secure-your-own-server/