React and Flask Stacks for Docker on the Raspberry Pi

Jim Ladd

January 12, 2021

SOFWERX

Contents

Introduction

The Clockwerx project started with the desire to program a set of wall mounted clocks via a web application. After a false start, the architecture was refactored to include a web application based on a React stack to provide a user interface. A web service based on a Flask stack was selected to provide the interface to the clock display mechanism. Both of these services are embedded in Docker containers and hosted on a Raspberry Pi computer. This document presents an overview of the architecture and several tips and details for building out a similar system.

Background

SOFWERX is a non-profit organization whose mission is to accelerate the evolution of the Warfighter through technology discovery, engagement, development, and transition. One of our strengths is the wide range of events that are hosted by our Events Team. One of the requests from the team was to install clocks in each of our conference rooms and allow an Event Coordinator to program the clocks to be a countdown timer. This mode, jokingly referred to as "speed dating", allows the attendees in each meeting room to know the amount of time remaining in their current session.

This project, internally called Clockwerx, allows a single user to program one or more of the clocks located throughout the facility. The system boundary of this project is shown below:

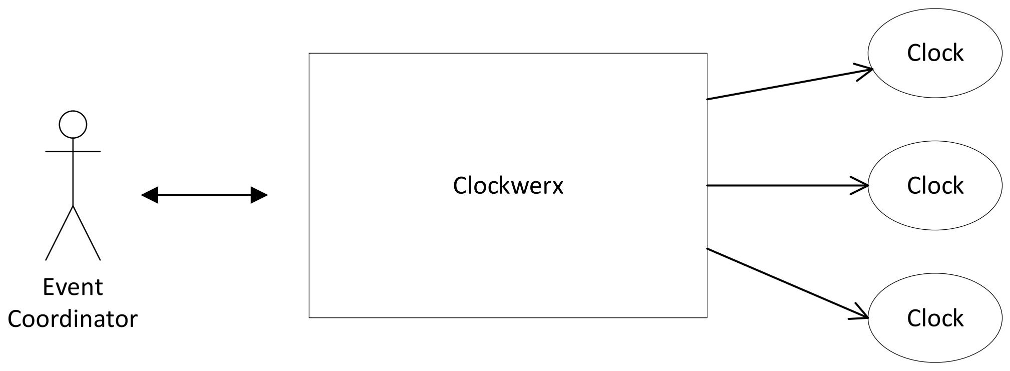

Figure 1 – System boundary

In order to keep this project as simple as possible, one of the few requirements was that there is a single user at a time. Clockwerx did not have to deal with resource contention issues or race conditions from multiple users. A design decision was made early in the project lifecycle that the "clock" would consist of an inexpensive, "dumb" display unit that was compatible with an infrared (IR) remote control similar to a TV. Each clock would be controlled programmatically by an IR interface installed on a Raspberry Pi (RPi) single board computer. By adding a Raspberry Pi to the architecture, the "dumb" unit could hopefully become a "smart" device. This smart device also had the potential to be controlled remotely via the Raspberry Pi wireless or wired Ethernet ports. Additionally, the user could program one or more of the clocks with a single user interface instead of accessing each clock individually. The concept of the architecture at this point is depicted in the following diagram.

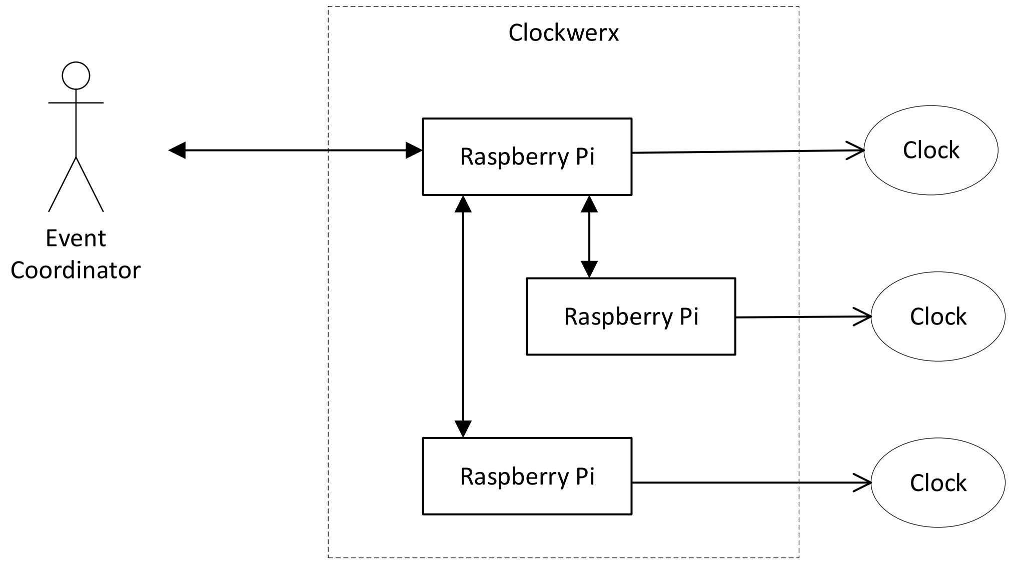

Figure 2 - A single Raspberry Pi controls a single clock via an infrared interface

The next step in the design process involved come unconventional yet interesting decisions. An architecture following traditional patterns would designate one of the Raspberry Pi as a "controller" and the other RPis as "responders". The controller RPi would be the one that the Event Coordinator would use to program all of the available clocks. The controller would have knowledge of the responders and how to communicate with them. While the controller/responder pattern is straightforward and somewhat intuitive, it does have some weaknesses. The deployment/configuration of the controller is now different than the responders. Also, there is a single point of failure inherent in a pure implementation. If the unit acting as the controller fails, another RPi must be reconfigured to fulfill that role.

A different approach leverages the single user requirement. Since resource conflicts are not a primary concern, each RPi can be configured the same and be used to control all of the other devices in the system. This technique does not rely on the Event Coordinator logging into the designated controller. The user may access any clock via the user interface hosted on any RPi.

To further encourage this direction of uniformity, the software on the RPi is partitioned into two sub-systems. The browser-based user interface software (Web App) is based on a NodeJS / React / Apache stack. It is responsible for providing a web application that allows the user to program one or more of the clocks. From the browser, it communicates with the web service that interacts with the clock display on the RPi via the infrared interface. This web service (Web Service) is based on a Python / Flask / Apache stack. A diagram of the two sub-systems is shown below:

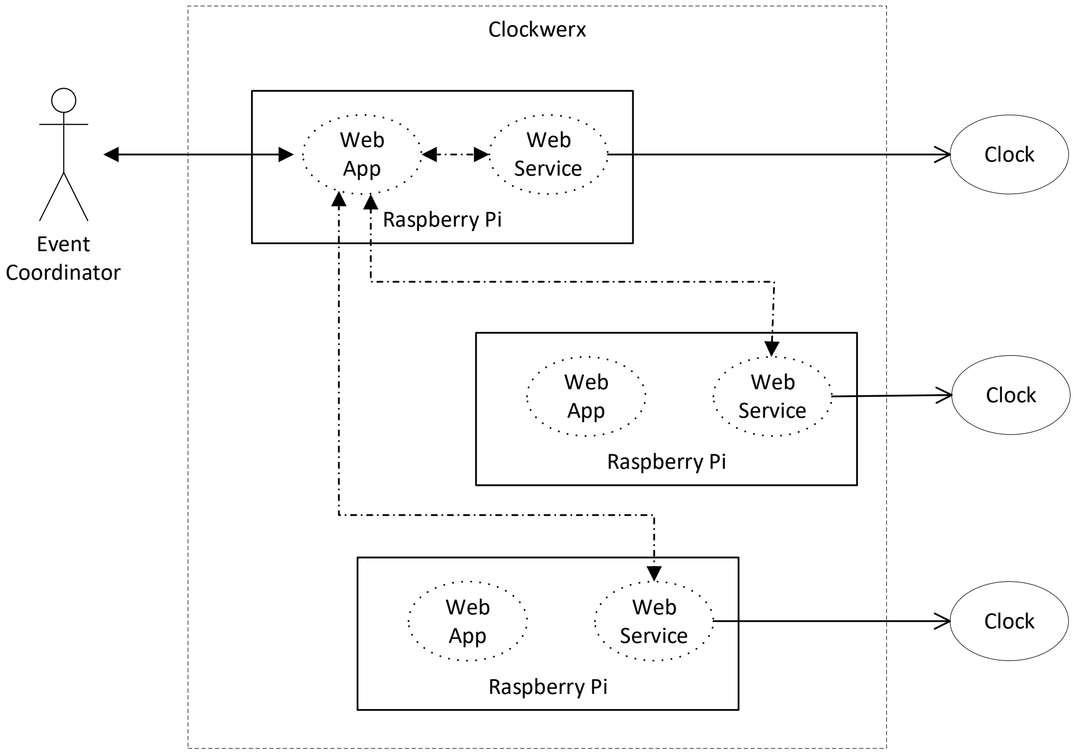

Figure 3 - Web app and web service interaction

Web App Stack

The Web App stack consists of a full web client environment that is contained in a Docker image and executed on a Raspberry Pi. This stack leverages NodeJS with React and Apache. It communicates with the Clockwerx Web Service via the Axios software. A graphical representation of this stack is shown in the following diagram:

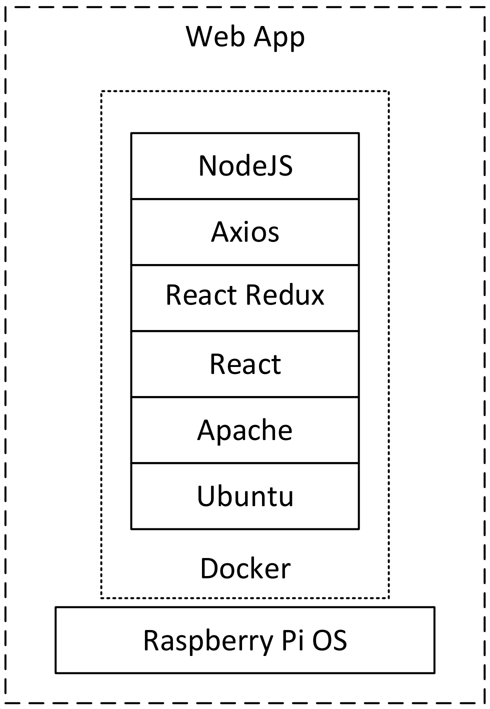

Figure 4 - Web application stack

The source artifacts are located in a GitHub repository located at:

https://github.com/sofwerx/clockwerx-web

The build the Docker image, a version of a simple command can be used:

```
docker build -t clockwerx-web .
```

The critical file is the Dockerfile located at the base of the project. For convenience, it is presented in Appendix A of this document. This file has the instructions for the installation and configuration for the Web App image. The first noteworthy item of the Dockerfile is that a “build” image is used in order to reduce the memory size of the final image. As the different software packages are installed, the Dockerfile build process will copy configuration files from the */config* directory to the appropriate location in the file system of the image.

One of the issues encountered during the migration of the code to a Docker container was a JavaScript memory limitation. Since the Docker image was constructed only for the production deployment, the following two configuration lines were added before the npm command to work around the memory issue.

```
RUN set NODE_OPTIONS="--max-old-space-size=4096"
ENV GENERATE_SOURCEMAP=false
RUN npm run build
```

Toward the end of the Dockerfile, the instructions will copy the required artifacts from the "build" image to the "deploy" image. The last step is to provide the *supervisord* server with the configuration file. This service facilitates starting the Apache server. The configuration file is located at:

```
config/supervisor.conf
```

The contents of this file are shown below. This file sets a few configuration properties and starts the Apache 2 web container. The remainder of the startup sequence is described in the ***Web Service Stack*** section of this document.

```
[supervisord]
logfile=/var/log/supervisor/supervisord.log
logfile_maxbytes=50MB
logfile_backups=5
loglevel=error
nodaemon=true

[program:apache2]
command=service apache2 start
```

One of the subtle but important challenges of this architecture is that the JavaScript executing in the browser must know the IP of the Web Service for all of the clocks. This includes the IP of its own host computer. Common techniques erroneously return the IP of the computer that is executing the JavaScript and NOT the IP of the computer that served the JavaScript code. A JavaScript method was created that executes a HTTP "GET" request for a known JSON file residing on the host computer. This specific code does not require the host IP and, by default, the host IP will be added to the outbound request by the HTTP library. The HTTP response to this request contains the host IP and port number in the "url" attribute. The URL component for the "fetch" request is simply the name of the file such as:

```
"/echo.json"
```

The "fetch" request returns a response with an attribute of:

http://192.168.1.215:4742/echo.json

This JavaScript function parses the IP and port number from this value. The IP values for all of the Web Services are maintained in a clock definition file (i.e. the *clockDefs.json* file) that is located in the Web Service. The Web App software requests this information from the Web Service. It uses the IP to direct HTTP requests with programming data to the desired clock.

Web Service Stack

The Web Service stack consists of a REST service that is contained in a Docker image and executed on a Raspberry Pi. This stack leverages Python with Flask and Apache. It receives a request from a client such as the Clockwerx Web App, executes the request, and returns a response with a status code along with any requested data. A graphical representation of this stack is shown in the following diagram:

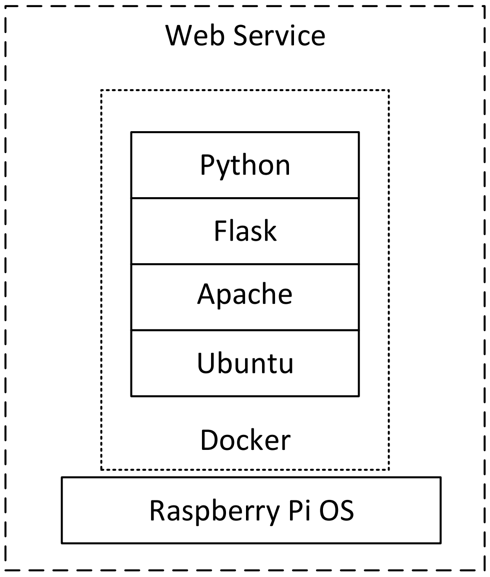

Figure 5 - Web service stack

The source artifacts are located in a GitHub repository located at:

https://github.com/sofwerx/clockwerx-ws

The build the Docker image, a version of a simple command can be used:

```
docker build -t clockwerx-ws .
```

Toward the end of the Dockerfile, the *supervisord* server is configured. This service facilitates starting the Web Service software at system boot. The configuration file is located at:

```
config/supervisor.conf
```

The contents of this file are shown below. This file sets a few configuration properties and starts the Apache 2 web container. The server for the infrared interface is also started.

```
[supervisord]
logfile=/var/log/supervisor/supervisord.log
logfile_maxbytes=50MB
logfile_backups=5
loglevel=error
nodaemon=true

[program:apache2]
command=service apache2 start

[program:lircd]
command=service lircd start
```

This solution uses the Docker Compose service to define, start, and stop both the Web App container and the Web Service container. The configuration is in the *docker-compose.yml* file located at the base directory of the *clockwerx-ws* repository. This document was changed from the production version since this repository is hosted in GitHub and not GitLab. Something similar can be constructed with GitHub. The *docker-compose.yml* file is presented in Appendix C of this document.

One of the concerns of this project was the process to deploy new versions of the software. The approach used for the production environment was to have a startup script that is executed at the end of the boot process for the Raspberry Pi. The */etc/rc.local* file was modified to execute the *startup.sh* file located in the clockwerx-ws project. This script stops the containers listed in the *docker-compose.yml* file, downloads the latest version of the docker images in the GitLab image repositories, and, finally, starts the containers via the *docker-compose.yml*. The startup.sh file is presented in Appendix E. A diagram of this process is shown below:

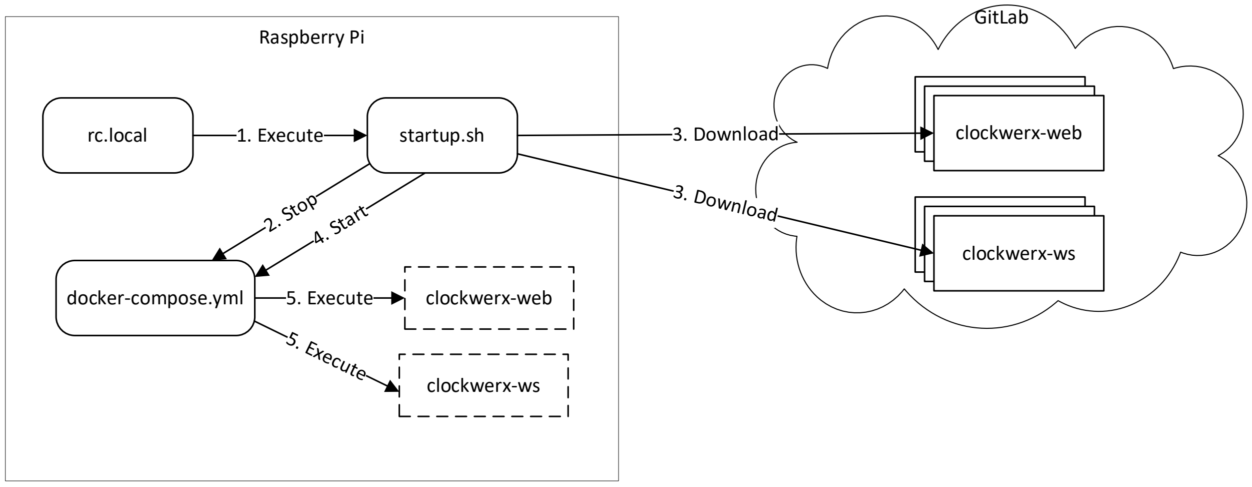

Figure 6 - Startup sequence

The user interface provides the Event Coordinator with the option to reboot one or more of the clocks. For each selected clock, the Web App sends a reboot request to the clock's Web Service. The Web Service then reboots the Raspberry Pi. During the reboot process, the latest versions of the Web App image and the Web Service image are downloaded and started. One subtle gotcha was that the code in the Web App must issue the reboot request to its own Web Service AFTER all of the other Web Services. Otherwise, its own Raspberry Pi would be rebooted and it would never send the request to the remaining Web Service instances.

Looking Back and Moving Forward

At first glance, the architecture for this project appears to be simple…just host a web app and a web service on a Raspberry Pi. Rarely is software architecture simple but it should always be straightforward. Getting the development versions of the two services installed and running was straightforward. This involved starting Web App and the Web Service from the command line without using Apache and without embedding them in Docker containers. This mode is highly recommended for the development of the core source code. Reducing the number of containers in the stack will greatly improve the speed of development.

As strongly as the recommendation is for a simplified development environment, a similarly strong recommendation is to develop the production environment as soon as possible. There was significant time and effort spent on getting the Apache server to work with the Flask server.   Another large time sink was the JavaScript memory limitation during the Docker build process for the Web App. These would have made major negative impacts to the timeline if the production environment was not developed early in the lifecycle.

Appendix A - Web App Dockerfile

This section presents the Dockerfile for the *clockwerx-web* repository. To build the docker image, enter the following command at the base of the project:

```
    docker build -t clockwerx-web .
```

```
# Configure the"build" image
FROM ubuntu:18.04 AS BUILD_IMAGE

# Get the updates
RUN apt-get update -y

#COPY set working directory
RUN mkdir -p /app
WORKDIR /app

# add `/app/node_modules/.bin` to $PATH
ENV PATH /app/node_modules/.bin:$PATH

RUN apt-get install nodejs -y
RUN apt-get install -y npm
RUN npm install -g npm@latest

COPY package.json ./
COPY .env ./
COPY echo.json ./

RUN npm install

RUN mkdir -p ./public
COPY public ./public

RUN mkdir -p ./src
COPY src ./src

RUN set NODE_OPTIONS="--max-old-space-size=4096"
ENV GENERATE_SOURCEMAP=false
RUN npm run build

# Now configure the executable image
FROM ubuntu:18.04

# Get the updates
RUN apt-get update -y

# Install and configure Apache
RUN apt-get install apache2 -y

WORKDIR /etc/apache2
RUN a2enmod headers
COPY config/apache2.conf .
COPY config/ports.conf .

WORKDIR /etc/apache2/sites-available
COPY config/clockwerxWeb.conf .
```

```
WORKDIR /var/www
RUN mkdir -p clockwerxWeb

WORKDIR /var/www/clockwerxWeb
RUN chmod -R 777 .
RUN mkdir -p logs
RUN chmod 777 logs

WORKDIR /etc/apache2/sites-available
RUN a2dissite 000-default
RUN a2ensite clockwerxWeb.conf

RUN apt-get install supervisor -y
RUN mkdir -p /var/log/supervisor
RUN mkdir -p /etc/supervisor/conf.d
COPY config/supervisor.conf /etc/conf.d/supervisor.conf

# Copy the build artifacts to the deployment folder
WORKDIR /
COPY --from=BUILD_IMAGE /app/build /var/www/clockwerxWeb


WORKDIR /
RUN alias python=python3
CMD ["supervisord", "-c", "/etc/conf.d/supervisor.conf"]
```

Appendix B - Web Service Dockerfile

This section presents the Dockerfile for the *clockwerx-ws* repository.   To build the docker image, enter the following command at the base of the project:

```
docker build -t clockwerx-ws .
```

The contents of the Dockerfile is:

```
FROM ubuntu:18.04

Run apt-get update -y

Run apt-get install lirc -y

Run apt-get install python-pip -y

Run pip install flask

Run apt-get -y install openssh-client

Run apt-get -y install sshpass

Run apt-get install apache2 -y

Run apt-get install libapache2-mod-wsgi -y

WORKDIR /etc/apache2

Run a2enmod headers

COPY conf/apache2.conf .

COPY conf/ports.conf .

WORKDIR /etc/apache2/sites-available

COPY conf/clockwerxWS.conf .

WORKDIR /var/www

Run mkdir -p clockwerxWS

WORKDIR /var/www/clockwerxWS

COPY app app

COPY conf/clockwerxWS.wsgi .

Run mkdir -p conf

COPY conf/clockDefs.json conf/

run mkdir -p logs

Run chmod 777 logs
```

```
WORKDIR  /etc/apache2/sites-available

Run a2dissite 000-default

Run a2ensite clockwerxWS.conf

Run apt-get install supervisor -y

Run mkdir -p /var/log/supervisor

Run mkdir -p /etc/supervisor/conf.d

COPY conf/supervisor.conf /etc/conf.d/supervisor.conf

WORKDIR /

CMD ["supervisord", "-c", "/etc/conf.d/supervisor.conf"]
```

Appendix C – docker-compose.yml

This section presents the docker-compose.yml file used on the Raspberry Pi. The source location is in the https://github.com/sofwerx/clockwerx-ws repository. Please note that the production version of this Clockwerx system is maintained in GitLab and not GitHub. The production docker-compose.yml file references the image in the GitLab repository. When the artifacts were migrated to GitHub for public consumption, the image names where changed to clockwerx-web and clockwerx-ws.

```
version: "3"
services:
        website:
                container_name: clockwerx-web
#                image: registry.gitlab.com/swxadmin/clockwerx-web
            image: clockwerx-web
                networks:
                        net:
                                ipv4_address: 172.4.11.2
                ports:
                        - "4742:80"
                volumes:
                        -
/var/log/clockwerx/clockwerxWeb:/var/www/clockwerxWeb/logs
                restart: always
                depends_on:
                        - web_service

        web_service:
                container_name: clockwerx-ws
#                image: registry.gitlab.com/swxadmin/clockwerx-ws
            image: clockwerx-ws
                networks:
                        net:
                                ipv4_address: 172.4.11.3
                ports:
                        - "4743:4743"
                devices:
                        - /dev/lirc0
                volumes:
                        - /etc/lirc:/etc/lirc
                        - /etc/modules:/etc/modules
                        - /etc/localtime:/etc/localtime:ro
                        -
/var/log/clockwerx/clockwerxWS:/var/www/clockwerxWS/logs
                restart: always

networks:
        net:
                ipam:
                        driver: default
                        config:
                                - subnet: "172.4.11.0/24"
```

Appendix D – JavaScript To Determine Host IP And Port

This section presents a JavaScript function that determines the IP and port number of the host computer. The function uses the "fetch" feature to retrieve a known file on the host computer. The HTTP response to this request contains the host IP and port number in the "url" attribute. An example of this attribute is:

http://192.168.1.215:4742/echo.json

```
async getHostIP() {
    console.log('Entering getHostIP');
    //var hostIP = "";
    var hostPort = "";
    try {
      var response = await fetch("/echo.json", {
        method: "get",
        mode: 'no-cors',
        headers: {
          'Content-Type': 'application/json',
          'Accept': 'application/json'
        }
      });

      var url = response.url;
      console.log('url = ' + url);
      this.state.hostIP = url.split('/')[2].split(':')[0];
      hostPort = url.split('/')[2].split(':')[1];

      console.log("hostIP=" + this.state.hostIP + " hostPort=" + hostPort);
    }
    catch (error) {
      console.log(error);
    }

    console.log('Exiting getHostIP');
    console.log("host ip: " + this.state.hostIP);
    return this.state.hostIP;
  };
```

Appendix E – startup.sh

This section presents the startup.sh file that is used during the system boot sequence. It stops the Docker containers, downloads the latest images from a GitLab repository, and then starts the containers via the docker-compose.yml file.

```
#!/bin/bash

grep -Fxq "swx_pi ALL=(ALL) NOPASSWD: /sbin/reboot, /sbin/poweroff,
/sbin/shutdown" /etc/sudoers || echo 'swx_pi    ALL=(ALL) NOPASSWD:
/sbin/reboot, /sbin/poweroff, /sbin/shutdown' >> /etc/sudoers

/usr/local/bin/docker-compose -f /home/swx_pi/docker-compose.yml down

docker login -u gitlab+deploy-token-1300241 -p <GITLAB PASSWORD>
registry.gitlab.com/swxadmin/clockwerx-ws

docker pull registry.gitlab.com/swxadmin/clockwerx-ws:native

docker logout

docker login -u gitlab+deploy-token-1300121 -p <GITLAB PASSWORD>
registry.gitlab.com/swxadmin/clockwerx-web

docker pull registry.gitlab.com/swxadmin/clockwerx-web:native

docker logout

docker image prune -f

/usr/local/bin/docker-compose -f /home/swx_pi/docker-compose.yml up -d
```