Osmotic Computing and Dealing With Back-pressure

Travis Higgins

May 2022

Computer Science MComp G405

Supervisor: Prof Raj Ranjan

Final Word Count: 100059

Abstract

The aim of this dissertation is "to evaluate the effectiveness of different methods to deal with backpressure in an osmotic computing pipeline". Osmotic computing “enables the automatic deployment of microservices that are composed and interconnected over both edge and cloud infrastructures”(Villari *et al.*, 2016). In this project an IoT-Edge-Cloud pipeline is created by using the Urban Observatory API, my PC, three Raspberry Pis and an AWS EC2 instance. One of the Raspberry Pis to balance the data to the other two where it is processed and then sent to the cloud. Experiments are performed to compare the least connections, round robin, and weighted round robin load balancing algorithms. The hypothesis is that the least connection algorithm will be the most efficient algorithm. Metrics are compared such as the time taken from when the data was sent from the load balancer until it was received in the cloud, and Grafana graphs showing data such as CPU usage to compare the different experiments. Upon evaluation, the hypothesis is disproven and the least connections algorithm only generally performed better in pipelines where different CPU constraints were applied to containers and the respective weights were not applied to in the weighted round-robin algorithm to compensate for this. It can be concluded that although the least connections algorithm did not always perform the best, it is the best algorithm to deploy easily if the programmer does not know the processing power of the different edge devices.

Declaration

"I declare that this dissertation represents my own work except where otherwise stated."

Aknowledgements

I would like to thank my supervisor Professor Raj Ranjan for supervising me on this project and also the P.H.D student Stanly Wilson Palathingal, who has provided support.

Table of Contents

1 Introduction **6**
1.2 The problem 6
1.3 The rationale 6
1.4 Aim and Objectives 7
1.5 Changes Since the Initial Proposal 7

2 Background Review **9**

3 Implementation **11**
3.1 Overview of the architecture of the project 11
3.2 Hardware, Technologies and Software Used In the Project 13
3.3 Issues encountered during the initial configuration of the development environment and how they were overcome 16
3,4 The Development Journey 17
3.5 Implementing container metrics monitoring 19
The SQL Database and Calculating Time Metrics 21

4 Results and Evaluation **22**
4.1 Why are these load balancing algorithms being compared? 22
4. 2 Rationale behind hypothesis 22
4.3 Limitations of the least connections load balancing algorithm 22
4.4 Why CPU Constraints are used in the expeirments 23
4. 5 Analysing the SQL Database Metrics 23
4.6 Analysing the Grafana Metrics 27
4.7 Evaluating results against the hypothesis 29

5 Conclusion **30**
5.1 Meeting the objectives 30
5.2 Justification for omission of objectives 4 and 5 31
5.3 Suggestions for Future Work 31

References **32**

Appendix **34**
Grafana Graph Experiments Results 34
SQL Database Results 43

1 Introduction

1.1 The context

Osmotic computing “enables the automatic deployment of microservices that are composed and interconnected over both edge and cloud infrastructures” (Villari *et al.*, 2016).
The Internet of Things is an ever growing field that is used in many different types of industries, such as healthcare and finance, where basic calculations on the data need to be processed in real time (and the latency caused by sending the data to the cloud, would be impractical), but often complex, resource intensive algorithms also need to be performed on large volumes of data, which often cannot feasibly be done through edge computing due to resource limitations. I believe that osmotic computing is the way forward to processing data more efficiently, instead of using just an edge computing or a cloud centric Internet of Things programming model, as this reaps the benefits of both models.

1.2 The problem

Backpressure is the CPU load or queue length, and can be defined as “Resistance or force opposing the desired flow of data through software” (Phelps, 2019).
Osmotic data flow can be affected by three main conditions, which can cause backpressure: limited downstream bandwidth between Layers (which can be due to a change in the network or poor network conditions), edge device stability (which can be caused by a hardware fault or load saturation), changes in IoT Devices Publishing to Edge Layer (which can be caused change in the number of IoT devices or a change in the frequency).

I would like to investigate how to implement measures to mitigate the effect that back-pressure has on the system, therefore making an osmotic computing pipeline more efficient and able to cope with large volumes of data. This is something which is essential if osmotic computing becomes the main paradigm used in an Internet of Things and cloud computing workflow.

1.3 The rationale

For this project, I will emulate an osmotic computing pipeline by using the urban observatory API as my Internet of Things layer from which I will read air pollution data, three Raspberry Pi’s will be used as my edge devices and AWS will be used for the cloud data centre. I will then implement different load balancing algorithms into the edge layer and compare how effective each of these algorithms are when the edge layer is overloaded with data and therefore subjected to back-pressure. I will use one of my Raspberry Pi’s for load balancing and the other two for processing the data. I will first save the data from the Urban Observatory API to a CSV file inside a linux virtual machine running on a windows computer. I will then send this data from the virtual machine to the first Raspberry Pi via the MQTT protocol which uses a publish and subscribe model. The subscribe code runs on the Raspberry Pi and the publish data runs on the virtual machine. Both the subscribe code and publish code have been written in Python. When the air pollution data is received on the Raspberry Pi, this sends this data to the other two Raspberry Pi’s for processing before being sent to the cloud, for storage in an SQL database.

I will store up a backlog of data from the Urban Observatory API in a CSV file on a virtual machine so there is enough data to cause a build up of back-pressure when this is sent to the first Raspberry Pi. The first Raspberry Pi then has to use load balancing algorithms to forward the data to the other two Raspberry Pis. On the Raspberry Pi's I will perform some simple calculations on the data, which will be calculating the mean average of the data and the range of the data. The three load balancing algorithms I will investigate the effectiveness of are least connections, round-robin and weighted round robin. I will mainly be investigating the effect of using different load balancing algorithms to distribute the data for processing between the other two Raspberry Pi's from the load balancer.

1.4 Aim and Objectives

Aim: To evaluate the effectiveness of different methods to deal with backpressure in an osmotic computing pipeline.

Objectives

1. To use the Urban Observatory API (IoT layer), with the Raspberry Pis acting as an edge device and using AWS for the cloud to emulate the data flow of an osmotic computing pipeline.
Explanation: This is necessary so that I can implement load balancing into the edge layer in order to evaluate the effectiveness of different methods and algorithms.
2. To investigate and evaluate the different algorithms that can be used for load balancing.
Explanation: An efficient load balancing algorithm should mitigate the effect that back pressure has on a system. This therefore is worthwhile researching as it will assist me in my aim of minimising back pressure in an osmotic computing pipeline.
3. To implement an efficient load balancing algorithm into the edge layer of an osmotic computing pipeline.
Explanation: I can compare how implementing different algorithms, such as round robin or least connections affect how well the edge layer copes with a high volume of data.

1.5 Changes Since the Initial Proposal

Since the proposal was submitted, the hypothesis has been formed. The hypothesis is that the least connections algorithm will be the most efficient load balancing algorithm when compared with round robin and weighted round robin. This provides a direct focus point which is discussed and evaluated in the results and evaluation section of this document. Furthermore, in the initial proposal there were 5 listed objectives, the last two objectives which have been omitted are listed below:

4. To Investigate, evaluate and compare means of adaptive data flow control.
Explanation: If the CPU load (backpressure) is too high then an algorithm such as the token bucket algorithm, or leaky bucket algorithm can be used in the edge gatekeeper to limit the forwarding rate. This will allow the edge device to be able to work through processing the backlog of data that has built up while minimising the chances of the edge device crashing.

5. To implement adaptive data flow control into an osmotic computing pipeline which is effective at minimising back-pressure.
Explanation: I will be able to evaluate how effective different methods of adaptive data flow control are at coping with high volumes of data and explore the benefits and drawbacks of these methods.

The decision to omit these objectives from the dissertation was taken because, as the project developed, it was decided that the main focus of the dissertation would only be on analysing and comparing load balancing algorithms. Experiments are performed to form conclusions as to which algorithm processes the data most effectively when subjected to different scenarios, such as varying docker CPU constraints, and therefore which algorithm is the most effective at mitigating the impact of back-pressure in different system configurations. By omitting objectives 4 and 5, more experiments can be performed and more data can be collected in the hope of being able to form more conclusive evaluation and comparison between these algorithms. These experiments are also used to evaluate the validity of the hypothesis.

2 Background Review

Here I will talk about the scientific papers I have read and their relevance to the dissertation. The first paper I read was “Osmotic Computing a New Paradigm for Edge/Cloud Integration”. I found this paper very useful as it gave me an overview of what osmotic computing is and helped me to decide that this is something I want to investigate because it is my belief that the osmotic computing model could be the future of IoT and cloud computing. This is because the osmotic computing model can “decompose applications into microservices and perform dynamic tailoring of microservices in smart environments exploiting resources in edge and cloud infrastructures” (Villari *et al.*, 2016). This means that with the osmotic computing model we are able to reap the benefits of both the edge computing model and the cloud computing model.

The second paper I read was “Osmotic Flow: Osmotic Computing + IoT Workflow”(Nardelli *et al.*, 2017). I found that this paper provides a good amount of detail on the actual components in an osmotic flow model (such as a data source, sink, transformation function, contract, osmotic resource manager, universal stream repository, node manager, and worker). This paper provides a more in-depth understanding of how the osmotic flow model is actually constructed and therefore will help me to achieve objective 1.

I also read a scientific paper titled “Performance Analysis of Load Balancing Algorithms” (Sharma, Singh and Sharma, 2008). This paper helps with objectives 2, and 3 as it gives examples of both static algorithms such as round robin or the central manager algorithm and dynamic load balancing algorithms such as the central queue algorithm or local queue algorithm and explains how these algorithms operate. The round robin algorithm explained in this paper has been incorporated into the project implementation of this project. This paper also gives insight into how the effectiveness of different load balancing algorithms can be compared. In this paper the parameters they used to evaluate the performance of the different load balancing algorithms are: overload rejection, fault tolerant, forecasting accuracy, stability, if they are centralised or decentralised, nature of load balancing algorithms, cooperative, process migration, and resource utilisation. This is an aspect that has been considered for the implementation in the project for this dissertation, as docker container metrics, and time metrics such as the average time taken from when a packet is sent from the load balancer and received in the cloud. The paper has also provided a useful definition of load balancing as “the process of improving the performance of a parallel and distributed system through a redistribution of load among the processors”.

Another paper I read was “Osmotic Message-Oriented Middleware for the Internet of Things”(Rausch, Dustdar and Ranjan, 2018). This paper explains what message-oriented middleware is and why it is important and is used in osmotic computing. This highlights the role that message oriented middleware has in osmotic computing and how it could be used in my project. After some further research, I decided the best way to send data from the virtual machine on my pc to the load balancer Raspberry Pi and from this Raspberry Pi to the other Raspberry Pis was using the Eclipse Mosquitto open source message broker that implements the MQTT protocol (which is a MoM protocol). The MQTT protocol has been used because MQTT works using the publish and subscribe model which makes it very easy

to send the data from one device to another (the sending device publishes the data, assigned to a topic, and the receiving device simply has to subscribe to this topic on the relevant port and ip address). Mosquitto is the most popular MQTT broker, and is well documented, hence why I chose to use this.

The final paper I read was “Modeling and Emulation of an Osmotic Computing Ecosystem using OsmoticToolkit”(Buzachis *et al.*, 2021). This paper explains how to emulate an entire osmotic computing ecosystem. Therefore, I feel that this paper provides sufficient information in regards to how I will create the osmotic computing pipeline (objective 1).

3 Implementation

This implementation section explains what was done and how.

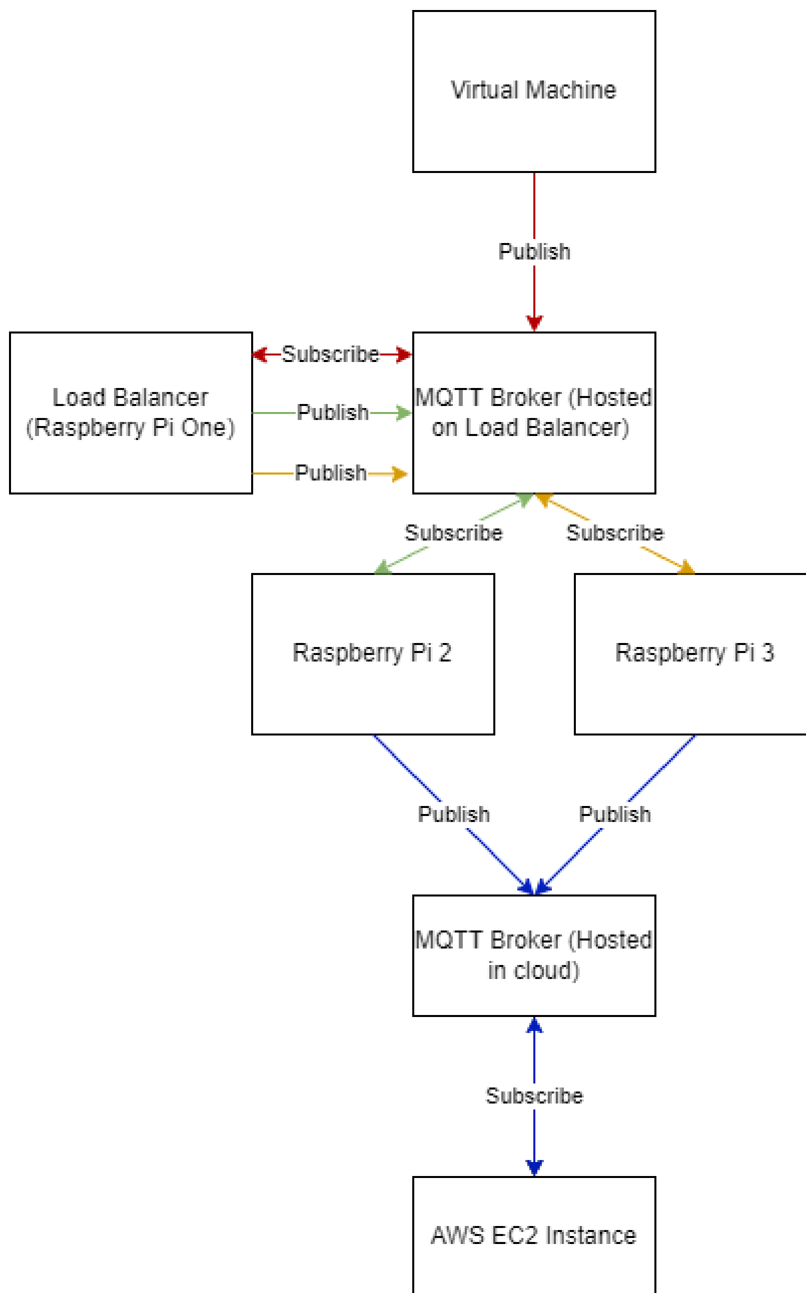


Figure 1: Diagram to show the flow of data using the publish and subscribe model

3.1 Overview of the architecture of the project

Throughout this document, the three Raspberry Pis have been referred to as Raspberry Pi one, Raspberry Pi two, and Raspberry Pi three respectively. Raspberry Pi one is used as a load balancer so is also interchangeably referred to as this. Raspberry Pi two and three are used to perform basic calculations on the data packets they have been sent.

Figure 1 illustrates how data packets are sent through the system across the system using the MQTT protocol. The diagram has been colour coded according to topics so that an arrow that is publishing to a topic and an arrow that is subscribing to the same topic are of the same colour.

The MQTT protocol has been implemented using the Mosquitto MQTT broker, to send the data packets from the virtual machine to the load balancer, and then from here to the other two Raspberry Pis and from them to the cloud. There is some Python code on the virtual machine called “publish.py”, which reads data from a CSV file and line by line this data is sent across to the mosquitto broker via MQTT. The data was published under the topic of “file_subscribe” so that this data could easily be subscribed to under the same topic name inside some python code on Raspberry Pi one. At first the code just printed each line of the CSV file that was received. The data was sent from publish.py using a sleep timer of 1 second between sending each message initially. This was done to make it very easy to check that the correct data was being sent across and that the application was functioning as intended. Using the sleep timer allowed the functionality of the round robin, and weighted round robin load balancing algorithms to be easily tested visually by observing if the payloads are sent to the Raspberry Pis in the intended order and ratios.

3.2 Hardware, Technologies and Software Used In the Project

For this project all of the code was run inside linux (either on a Raspberry Pi or on a virtual machine) because linux is the preferred platform for using the Mosquitto MQTT broker. This made the system easier to implement and less error prone.

After the initial configuration, all of the project's development was done using one windows computer. This was made possible by connecting Putty with the three Raspberry Pis via SSH, so that the terminal of the Raspberry Pis could be remotely accessed. SSH was used to interact with the three Raspberry Pis because this was much more practical than plugging the keyboard, mouse and hdmi cable out of one Raspberry Pi and into another every single time some code needed to be edited or run. Using SSH made it a lot more practical to copy code from the VSCode to the Raspberry Pi this way. Initially VNC (which also uses SSH) was also used to interact with the GUI on the Raspberry Pis as well, but after using this for a while it became apparent that using the GUI was unnecessary as everything could be done faster via the command line interface.

When developing the code and Dockerfiles to go onto the Raspberry Pis, VSCode was used to first write out all of the code and then it was simply copied across to the Raspberry Pis by copying all the code and then pasting the contents into a file by using the nano command to edit the text within a file on the Raspberry Pi. VSCode was chosen for the development because it is a very light code editor in comparison to more traditional Integrated Development Environments that are used for python (such as PyCharm). With VSCode the user can simply install extensions and then code in most programming languages inside the VSCode editor. This would be of benefit if the project was taken further and another language was used, as the development could still be done using the same code editor.

To run the virtual machine a piece of software known as Virtual Box was used. VirtualBox was used because it is open source and free and recognised as one of the best virtualisation software tools available at the current time of writing. The Ubuntu distribution of linux was used on the virtual machine. The exact version of Ubuntu used was 20.04.4 LTS. There are several different distributions of linux available due to the nature of it being open source, however in this project Ubuntu was used because this is the most commonly used and most supported linux distribution there is and is the de facto standard distribution of linux most people will use unless they have a specific reason otherwise. This also means there is the most support for Ubuntu online when compared to other distributions so it should be easier to get help if any issues are encountered.

In this project the paho-mqtt client library was used to enable an easy way to publish and subscribe to MQTT broker in Python. MQTT implements three different Quality of Service constraints: QoS 0, QoS 1, QoS . In this project a Quality of Service value of 0 was used as the default as this means data can be sent and received faster, although receipt of the data is not guaranteed, as there is no handshake involved. In some systems a QoS 1 may be required (the data is received at least once), or QoS 2 (the data is always received exactly once).

This project was implemented using the mosquitto MQTT broker to communicate between all devices within the architecture of the project. MQTT is most commonly built on top of the TCP/IP stack. Mosquitto was used because it is lightweight and implements the publish and subscribe model which is, in the instance of this project, the most logical model to use because this allows data to be published directly from one device to another device but also multiple devices can publish to one device. This functionality has been utilised in this project by Raspberry Pi two and Three both publishing to the cloud under one topic so all the data is collated together.

As a result of how the publish and subscribe model publishes topics to the broker and subscribes to the topics from the broker, there is no direct connection between the publisher(s) and the subscriber(s), this means that systems that implement this kind of architecture are usually very easily scalable because it can be as simple as adding extra docker containers and publishing or subscribing to the relevant topic within them. For example, in the implementation of this project it would be very easy to add another docker container which publishes to the cloud because all the data in the cloud is received via subscribing to one topic called “subscribeawscloud”. This scalability is important because if future work is to be conducted using this project and the data needs to be load balanced across more nodes, then this should be easy with the current implementation used.

Shell scripts were also used in the development of this application in order to make the process of running the docker commands to stop cadvisor and prometheus containers, remove them and then recreate them again much faster than typing all of the commands manually. The cadvisor containers were stopped, removed and rebuilt instead of simply being restarted because this is best practice and also ensured all the comparisons were fair as then in each experiment, every cadvisor container was being run under exactly the same conditions.

The Python 3.8 image was used to run the code inside docker containers because dynamic typing (a key difference between Python and a lot of other programming languages) makes the development faster (but this can potentially mean there is a higher chance of errors due to the wrong type casting in larger applications). Another benefit of using Python is that it is installed on Ubuntu and the Raspberry Pi operating system by default so if the code is not being run in the container during the development of the application or while testing, this means there is no additional installation required to get the application to run.

The Raspberry Pis used as the edge devices in this project are the Raspberry Pi Three Model B+ w are used as the edge devices in this project because they are resource constrained, and therefore representative of most edge devices found in real world applications. In this system Amazon Web Services (AWS) is being used to host the cloud instance. AWS was used to host the cloud element of the project because it is reliable and a leading cloud provider.

A MYSQL database was used to store all the data received in the cloud. The MySQL database was run inside a docker container. A Python library called MySQL connector was used to connect to the MySQL database from within the Python code (and hence be able to execute MySQL queries). The MySQL database was able to connect to the code receiving the data from the Raspberry Pis because both images have been built in the same

docker-compose file, hence by default they run on the same docker network unless otherwise specified. There were issues encountered initially with connecting to the MySQL database from within the Python file. This was solved by adapting the code for a docker-compose file from a tutorial (Mothish, 2021).

In this project a few python libraries and modules were used. The libraries that were used are listed below, along with a justification to why and how they were used:

- Requests - This library is used to read data from API endpoints.
- Csv - This reader object from this module was imported so that this could be used to easily read each line of the csv file on the virtual machine one line at a time (by using a for loop), to then publish to the file_subscribe topic.
- Time - This module was used when testing the application so a sleep of one second could be used after each line of data was sent to the load balancer. This was done so it was easy to see if the load balancing algorithms and publish and subscribe mechanisms were working correctly.
- The Paho-mqtt client library made it possible to utilise the Mosquitto MQTT broker with Python code.

Docker containers were used to run the code on both the Raspberry Pis and the cloud because an osmotic computing pipeline uses microservices so by using docker containers this helps the application resemble this to help achieve objective one. The benefits of running applications inside containers is that all the code runs in an isolated environment so as long as the code runs correctly inside the container on the original machine, it is then very easy to then run this code on any other machine because the code running inside the container is not dependant on the dependencies installed on the machine, so therefore there cannot be any issues with incompatible dependency versions etc. This is something that is quite important to the architecture of my project, because the fact that all the code runs inside containers, means it will be very easy to add extra devices to the already existing architecture without any or minimal configuration problems. Having the ability to add cpu-constraints means it is very easy to make some edge devices perform less powerful for testing and evaluating purposes (especially for testing the weighted round robin algorithm). Docker containers were used over alternatives such as Podman, OpenVZ because docker is the de-facto standard, and the service for which there is the most online support for.

3.3 Issues encountered during the initial configuration of the development environment and how they were overcome

During the initial configuration of the environment of the project there were several issues, which meant this took significantly longer than expected. The details of the issues encountered and how they were solved or mitigated are explained below.

The project architecture involves installing Ubuntu onto a virtual machine in VirtualBox. Virtualbox is a hypervisor which can be used to run a virtual machine on a computer. Initially there were issues with the virtual machine freezing when installing the operating system for the first time, however once everything was installed and the virtual machine was running, the virtual machine was responsive and there were no issues with it freezing. There were also often issues with loading the virtual machine, after the Ubuntu image had been successfully installed on the machine. To mitigate the potential time constraints regularly dealing with this issue could impose on the project, the computer and virtual machine were always left running and not turned off during the development phase of the project.

There were further issues with the virtual machine as in an ideal configuration the developer should be able to either copy and paste files into the virtual machine from the host machine or be able to use a shared folder with the host machine. This would enable the developer to write the code locally on the host machine and then use the virtual machine only for running the code. Three methods to transfer the code from the local machine to the virtual machine were attempted:

- allowing bidirectional copy and pasting and drag and drop between the virtual machine and local machine
- using a usb flash drive
- creating a shared folder between the local machine and the virtual one.

With all the above methods, varying issues were encountered, the most common one being that in order to implement any of the above in Virtual Box, guest additions had to be installed. After installing the guest additions, the virtual machine would break and the virtual machine would need to be deleted and Ubuntu would have to be installed again on a new virtual machine. Trying to fix this issue was taking far too much development time so the decision was made that it is better to just simply do the development inside the virtual machine as the overhead of doing the development in a potentially laggy virtual machine, did not outweigh the extra time required to get a method of being able to share files between the two operating systems working.

3,4 The Development Journey

After the communication between the virtual machine and the load balancer was established, the next step was to implement communication from the load balancer to Raspberry Pi 2 and 3, and implement the round robin load balancing algorithm on the load balancer to send the data to the other two Raspberry Pis. The data was sent under different topics depending on which Raspberry Pi was the intended recipient, so each Raspberry Pi only needed to subscribe to its respective topic. In the load balancer code, there is a function called on_message_load_balance which decodes the message payload and then sends the decoded message to the round_robin function. This was done so a function for each different load balancing algorithm can be created, and then when running experiments, the load balancing function that is called is simply altered in the on_message_load_balance_function.

After the round robin load balancing algorithm was working over the MQTT protocol between the load balancer and Pi 2 and Pi 3, the decision was made to containerize all the code running on all of the Pis. When installing Docker onto the Raspberry Pis an error was encountered as shown below.

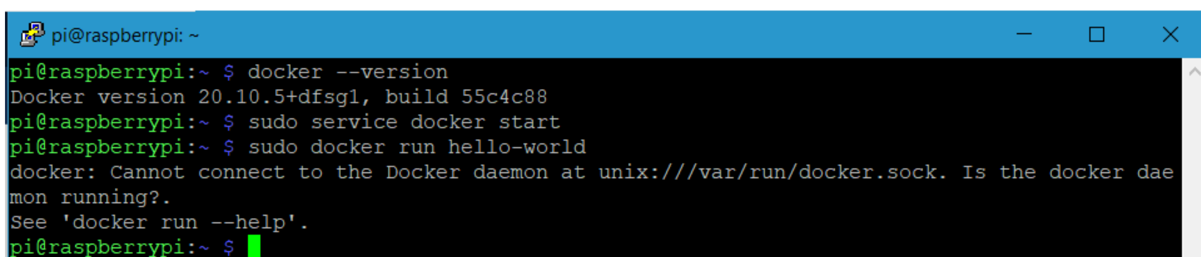


Figure 2: Error encountered with Docker

The cause of this issue was the installation process that was used to install docker as snap install used to install Docker on the first Raspberry Pi but this caused the error shown above. On Raspberry Pi 2 docker was installed using a script but the same error occurred. It was discovered that when installing docker using the “sudo apt install docker.io” command on Raspberry Pi 3, that the hello world test image could be run and then when the existing versions of docker were uninstalled from the other two Raspberry Pis and then reinstalled the “sudo apt install docker.io”, the test image could also be successful run on these machines as well. Getting the hello world image on docker working after installation was important because then if errors are encountered later down the line when trying to run docker containers, we can be confident that the error is with the code and not with the installation.

Once docker was correctly installed on all three Raspberry Pi’s the code could be easily containerised by adding a small docker file in the same directory as each of the Python files and then building and running the docker containers.

After this the weighted round robin algorithm function was implemented. This function distributes the payloads via a weighted round robin algorithm, where the weight of each

Raspberry Pi was specified in a dictionary as a parameter passed to the function, with the name of the topic being the key, and its respective weight being the value. The round robin and weighted round robin algorithms are shown below.

```python
# This function will implement the round robin load balancing algorithm to
# determine which Raspberry Pi to send the data to.
def round_robin(msg: str, topics : list):
    global current_index_round_robin
    topic_to_send_data = topics[current_index_round_robin]
    print(topics[current_index_round_robin] + " : " + msg)
    if current_index_round_robin >= len(topics)-1:
        current_index_round_robin = 0
    else:
        current_index_round_robin += 1
    publish_result(msg, topic_to_send_data)

# This function will implement the weighted round robin load balancing
# algorithm to determine which Raspberry Pi to send the data to.
def weighted_round_robin(msg : str, topics_and_weights: dict):
    global current_index_round_robin
    global num_of_iterations_current_weight
    topics_and_weights = {"Pi_two_subscribe" : 1, "Pi_three_subscribe": 5}
    topics = list(topics_and_weights)
    topic_to_send_data = topics[current_index_round_robin]
    print(topics[current_index_round_robin] + " : " + msg)
    if num_of_iterations_current_weight < topics_and_weights[topic_to_send_data] - 1:
        num_of_iterations_current_weight += 1
    else:
        num_of_iterations_current_weight = 0
        if current_index_round_robin >= len(topics)-1:
            current_index_round_robin = 0
        else:
            current_index_round_robin += 1
    publish_result(msg, topic_to_send_data)
```

Figure 3: Python code used to implement round robin and weighted round robin algorithms.

The next stage in the development process was to integrate the cloud into the current system. Using a PEM file and the relevant IP address an Amazon Web Services EC2 instance could be accessed through Putty to give access to the command line interface of the EC2 instance. Connecting Putty to the EC2 instance was done by following a tutorial (*How to Connect to your EC2 Instance using PuTTY V1.1*, no date). The EC2 instance was a Ubuntu machine.

Raspberry Pi 2 and Raspberry Pi 3 could be easily connected to the AWS EC2 instance by reusing the already existing code that implements the publish and subscribe model but by changing the IP address to that of the EC2 instance.

3.5 Implementing container metrics monitoring

The next stage in the development journey of the project was to implement a method of being able to analyse the docker containers and metrics such as their CPU usage. To do this, three pieces of software were used:

- cAdvisor - “provides container users an understanding of the resource usage and performance characteristics of their running containers”(*cadvisor: Analyzes resource usage and performance characteristics of running containers*, no date)
- Prometheus - “Prometheus is one of many open-source projects managed by the Cloud Native Computing Foundation (CNCF). It is monitoring software that integrates with a wide range of systems natively or through the use of plugins.”(*What is Prometheus and Why Should You Use It?*, 2020)
- Grafana - “Grafana is an open source solution for running data analytics, pulling up metrics that make sense of the massive amount of data & to monitor our apps with the help of cool customizable dashboards.”(Knoldus Inc., 2022)

One of the problems encountered with configuring this system was that the default google image for Cadvisor is not designed to run on ARM64 architecture, which is the architecture that the Raspberry Pi 3 Model B+ runs on which was used in this project. To resolve this issue, an unofficial version of the image which has been adapted for ARM64 was used instead (ZCube, no date).This docker image was then run on all three Raspberry Pi’s by executing the command found in the readme on the google cadvisor github page (*cadvisor: Analyzes resource usage and performance characteristics of running containers*, no date). A docker-compose.yml file was created to run on the virtual machine. The Docker compose file included images for Prometheus and Grafana. In the same directory there is also a prometheus.yml file. Inside the Prometheus.yml file the three targets are specified (the three cAdvisor containers running on the Raspberry Pi’s). When the command docker-compose up is run, this would then run the Prometheus and Grafana images.

When the ZCube cadvisor image was originally run, cAdvisor would run on the port 8080 and the interface was accessible through the web browser on the Raspberry Pi’s, however none of the running docker containers could be seen. To try and figure out what was causing this issue, the cadvisor image was run on Ubuntu on the virtual machine, to see if the same issue occurred there, but all the containers were visible when the image was run on Ubuntu. This indicated that the problem must be with the configuration of the operating system on the Raspberry Pis. After some research the cause of the issue was identified. The text “systemd.unified_cgroup_hierarchy=0 cgroup_enable=memory cgroup_memory=1”(*Which cgroup flags to add to cmdline.txt on 64bit Bullseye for Docker/Kubernetes - Raspberry Pi Forums*, no date) needed to be added to the /boot/cmdline.txt file in all of the Raspberry Pis and they all needed to be rebooted. This was an issue with the version of the Raspberry Pi operating system that was being used. After doing the reboot and then running cAdvisor again, all of the live container metrics could now be seen.

To see this data in Grafana, the set up was relatively simple. First the user must login (using the credentials which are specified in the docker-compose file), and then add prometheus as a source (by specifying that the port was on localhost:9090). A dashboard, found on the grafana website for docker containers (*Docker Container*, no date) was used and modified

so that the graphs are not stacked to enable better readability and to allow fairer comparisons. Although at this point the Grafana was working and the docker container metrics could be monitored, none of the data was being persisted so if the Grafana container was shut down. When the user logs back in again, Prometheus would have to be re-added as a source and there would be no saved dashboards, which would be time consuming and impractical to configure every time the Grafana image was run. To solve this volumes needed to be added to the docker compose file for the Grafana image as shown below. This was implemented by adapting the information for the Grafana docs by adapting the information provided on the Grafana docs (*Configure Grafana Docker image*, no date) for a docker compose file.

VirtualBox offers port forwarding, so this was used to map the ports for Prometheus and Grafana, on the virtual machine onto the same ports on the host machine (9090 and 3000 respectively). This was useful because the web browser on the virtual machine was sometimes slow to load in comparison to that on the host machine and made screenshotting the Grafana graphs for the experiments much easier as the virtual machine had a reduced screen size.

The SQL Database and Calculating Time Metrics

At various points in a messages journey from the virtual machine to the cloud, datetimes were added to the message so these can be used to calculate metrics to compare for different experiments. A datetime was added to the message, when it was sent from the VM, when it was sent from the load balancer, and when it was sent from the cloud. These datetimes have been used to calculate the network latency between the Raspberry Pi 2 or Raspberry Pi 3 and the cloud, the time taken from when the data is sent from the VM and when it is received in the cloud and also the time taken from when it is sent from the load balancer and then received in the cloud. An example calculation for the network latency from a Pi 2 or Pi 3 to the cloud is shown below:

Network Latency = Time Received in the cloud - Time sent from Pi

Saving the data inside a MYSQL database was beneficial (instead of, for example, just writing the data to a CSV file) because this allowed multiple different experiments to be run and each set of results to be stored in a separate table. The tables could easily be queried to retrieve the desired data. For example, the data was stored inside a database called air_pollution_data. To access the average metric for the time from load balancer to cloud for an experiment, all that needed to be run was the following commands:

- USE air_pollution_data;
- SELECT AVG(load_balancer_to_cloud) FROM example_experiment_data;

This made getting the averages needed to compare the results significantly faster and less error prone than manual methods.

4 Results and Evaluation

4.1 Why are these load balancing algorithms being compared?

The round robin, weighted round robin and least connections load balancing algorithms are being compared because this allows both static (round robin and weighted round robin) and dynamic load balancing algorithms (least connections) to be compared. By imposing CPU constraints on some of the docker containers, weights can be assigned with the weighted round robin algorithm in proportion to the CPU constraints to allow for a fair comparison between least connections and weighted round robin in an environment where not all nodes have the same processing power. This is something that can be common in Internet of Things programming models. A well known disadvantage of the round robin load balancing algorithm is its inability to efficiently accommodate for systems with different nodes of different processing power. Nonetheless, in systems where the processing power of the nodes is equal, the round robin algorithm has been known to perform well and this is why the first three experiments (which compare all three algorithms) do not have any docker constraints implemented.

4. 2 Rationale behind hypothesis

I have hypothesised that the least connections algorithm will be the most efficient load balancing algorithm when compared with round robin and weighted round robin, because with this algorithm the data packets will only be sent to the server with the least amount of active connections (data still being processed), therefore each data packet should theoretically only ever be sent to the server which is currently processing the least amount of packets. A limitation of the least connections load balancing algorithm is that a higher amount of processing is required because acknowledgements need to be sent from the Raspberry Pi 2 or Raspberry Pi 3 to the load balancer when each message has been processed, so that the load balancer can keep a record of how many active connections are one each Raspberry Pi.

4.3 Limitations of the least connections load balancing algorithm

Despite the fact that the round robin algorithm is the most simple load balancing algorithm that exists, in configurations where all of the servers have a similar processing power, it can be very effective and is by far the easiest to implement and it cpu power does not need to be wasted by receiving and sending acknowledgments that the data has been processed unlike with the implementation of the least connections load balancing algorithm. Another minor issue with the implementation of the least connections algorithm in this project is that sometimes a data packet will be sent to a server that does not truly have the least connections. For example, if server A has 150 active connections and server B has 149 active connections but has just completed processing two of the data packets resulting in their now only being 148 active connections on server A, if a packet was to come to the load balancer at this exact time then the data would be sent to server B despite this server actually having more active connections. This is because there is a small network latency delay between the acknowledgement being sent to the load balancer receiving this. Although this does mean the algorithm may not always perform 100 % optimally I believe this will

make negligible difference on the throughput which the system can process effectively as in this scenario both the servers are processing a very similar number of packets anyway. When the data is sent with a QoS 0 to the MQTT broker there will also be a small risk of the acknowledgement not being received could also have the potential to make the current connections counter in the load balancer inaccurate.

4.4 Why CPU Constraints are used in the expeirments

For the project Raspberry Pi 3s model 3 B+ Rev 1.3 have been used. These models of the Raspberry Pi use a quad core processor. As stated in the docker documentation (*Docker Documentation*, 2022) the command "--cpus" can be used to specify how many CPU cores to use (or even less than a whole core). This can be used in conjunction with the weighted round robin algorithm to be able to evaluate the algorithm's effectiveness. It is necessary to use CPU constraints for analysing the weighted round robin algorithm because it is designed to distribute different amount of packets to different nodes depending on their processing capabilities and because in this project all of the Raspberry Pis have the same specification and the same hardware, this allows the docker container to be constrained so that it can emulate a node with less processing power and slower hardware.

4. 5 Analysing the SQL Database Metrics

In these experiments, data metrics about the different experiments have been stored in the MySQL database. This evaluation uses the average of these three metrics from the MySQL database:

- Metric 1 - "Pi to cloud"
- Metric 2 - "Load balancer to cloud
- Metric 3 - "VM to cloud"

In the results shown in table 1, and table 2 weights refers to the weight assigned to each Raspberry Pi in the weighted round robin algorithm and CPU constraints refers to the CPU constraints which were applied to the docker containers performing the processing on each Raspberry Pi. The CPU constraints refers to the maximum number of CPU cores the docker container is allowed to use (the maximum is 4). For both CPU constraints and weights, they are listed in the same respective order (Raspberry Pi 2: Raspberry Pi 3). The experiments are listed below:

1. Round Robin - CPU constraints: none
2. Least Connections - CPU constraints: none
3. Weighted Round Robin - weights: 1:1 - CPU constraints: none
4. Weighted Round Robin - weights: 1:4 - CPU constraints 1:4
5. Weighted Round Robin - 1:16 CPU 0.25:4 CPU constraints: 0.25:4
6. Round Robin - CPU constraints: 1:4
7. Round Robin - CPU constraints: 0.25:4
8. Least Connections - CPU constraints: 1:4
9. Least Connections - CPU constraints: 0.25:4

Experiment Number	Pi to cloud (ms)	Load balancer to cloud (ms)	VM to cloud (ms)
Experiment 1	260	31000	31000
Experiment 2	254	37200	37200
Experiment 3	256	31000	31000
Experiment 4	258	38300	38300
Experiment 5	257	28600	28600
Experiment 6	254	39100	39100
Experiment 7	261	35100	35100
Experiment 8	251	44600	44500
Experiment 9	250	35000	35000

Table 1: The average results of the top 1000 rows for the experiments rounded to 3 s.f

Experiment Number	Pi to cloud (ms)	Load balancer to cloud (ms)	VM to cloud (ms)
Experiment 1	260	31000	31000
Experiment 2	254	37400	37400
Experiment 3	257	34300	34300
Experiment 4	257	40600	40600
Experiment 5	258	30700	30700
Experiment 6	254	35900	35900
Experiment 7	262	46400	46400
Experiment 8	251	46900	47000
Experiment 9	252	41800	41900

Table 2: The average results of the top all rows for the experiments rounded to 3 s.f

All of the above data has been rounded to 3 significant figures to allow a clearer comparison between the data (the unrounded values can be found in the appendices). Both the Grafana graphs and “pi_to_cloud” (network latency), “load_balancer_to_cloud” and “vm_to_cloud” metrics stored in the sql database, give useful insights. Unless stated otherwise the time metrics data that will be compared in this dissertation are that of the top 1000 rows shown in table 1. It can be argued that the SQL data is more valuable than the Grafana data as it allows a direct comparison of times taken. The network latency is a very useful piece of data

as knowing this variable we can check that all the data values are relatively close, and so an external factor (such as there being a lot of outgoing data from another PC on the network) has not drastically affected the network latency and therefore caused anomalies in the results.

All of the “pi_to_cloud” metrics for the top 1000 rows are between 250ms and 261ms, therefore there is negligible difference between them. If there was a higher variance, the network latency metric (“pi_to_cloud”) could be subtracted from the “VM_to_cloud” metric to ensure there is a fairer comparison between the different experiments. Two tables of data have been extracted from the MySQL database; the averages of all of the data and the average of the top 1000 results. During some experiments, socket timeout errors were experienced due to the high volumes of back-pressure which has resulted in some experiments having less rows in their SQL table than others . All of the experiments have at least 1000 entries so the average of the top 1000 rows for all three columns in every experiment has been recorded to allow a fairer comparison. Averages of all of the data have also been recorded.

Assumptions that were taken when conducting these experiments:

- All 3 Raspberry Pi’s have the same processing power and there is no hardware faults that affect performance (all of the Raspberry Pi’s are the same model)
- The rate at which data is sent from the virtual machine to the load balancer is approximately consistent throughout all experiments

Experiments 1, 2, and 3 (no CPU constraints)

Experiment 3 was undertaken as a control to compare with experiment one as, theoretically the results should be the same because when a weight of one is applied to both edge devices, this is essentially the same algorithm as round robin. The experiments produced the expected results and metrics 2 and 3 are both 31000ms for both experiments. Experiment 2 performed slower than 1 and 3 as it took 37200ms for metric 2 and 3 which is 62000ms more than experiments 1 and 3. This could be explained by the extra decisions that have to be made in the implementation of the least connections load balancing algorithm (for each packet of data sent to this algorithm between one and three conditional statements will have to be checked whereas with round robin there is only one if and one else statement). Furthermore, in the least connections algorithm, the load balancer has to subscribe to both the data coming from the virtual machine and also the acknowledgements for when a connection has closed. The extra processing required in this algorithm will potentially impact the speed at which the messages are allocated and sent to the respective Raspberry Pis, when compared to the other two algorithms.

Experiments 4, 6 and 8 (1: 4 CPU constraints)

As is to be expected, metrics 2 and 3 are faster in experiment 4 than they are in experiment 6. Metrics 2 and 3 are both 38300ms in experiment 4 and are both 39100ms for metrics 2 and 3 in experiment 6. This means there is a difference of 800ms. Surprisingly the metrics for experiment 8 were the slowest, with 44500ms for metrics 2 and 3, which is 5400ms slower than experiment 6 and 6200ms slower than experiment 8.

Experiments 5, 7 and 9 (0.25:4 CPU constraints)

Once again weighted round robin performed faster than round robin here, with metrics 2 and 3 being 6500ms faster on experiment 5 than experiment 7. Experiment 9 did perform better than experiment 7 by 100ms for both metrics 2 and 3. This is only a very marginal difference, and weighted round robin has outperformed least connections by 6400ms for metrics 2 and 3.

Experiments 1, 6, 7

Experiments 1, 5 and 7 are all round robin experiments. Experiment 1 performs the fastest which is to be expected because there are no CPU constraints, however, experiment 7 performs 4000ms faster than experiment 6. As the docker container on Raspberry Pi 2 has a quarter of the processing in experiment 7 than it does in experiment 6 and all the data packets are being evenly distributed by round robin, it must be concluded that this result is an anomaly. When the results of all of the data are compared (table 2), experiment 6 does in fact perform 10,500ms slower (46400-35900), which supports the notion that this result is an anomaly.

Experiments 2, 8, 9

Experiments 2, 8 and 9 are experiments that implement the least connections algorithm. Surprisingly, experiment 9 performed the best, followed by experiment 2 and then experiment 8 being the slowest when looking at the data in table 1. On the contrary to this, experiment 2 performs the fastest, followed by experiment 9 and then experiment 8 when we look at the averages for all of the data shown in table 2. I think the difference in results between these two datasets is due to the container on Raspberry Pi 2 crashing 3 minutes before the container on Raspberry Pi three stopped. This can be seen in the grafana graph for experiment 8 found in the appendix.

4.6 Analysing the Grafana Metrics

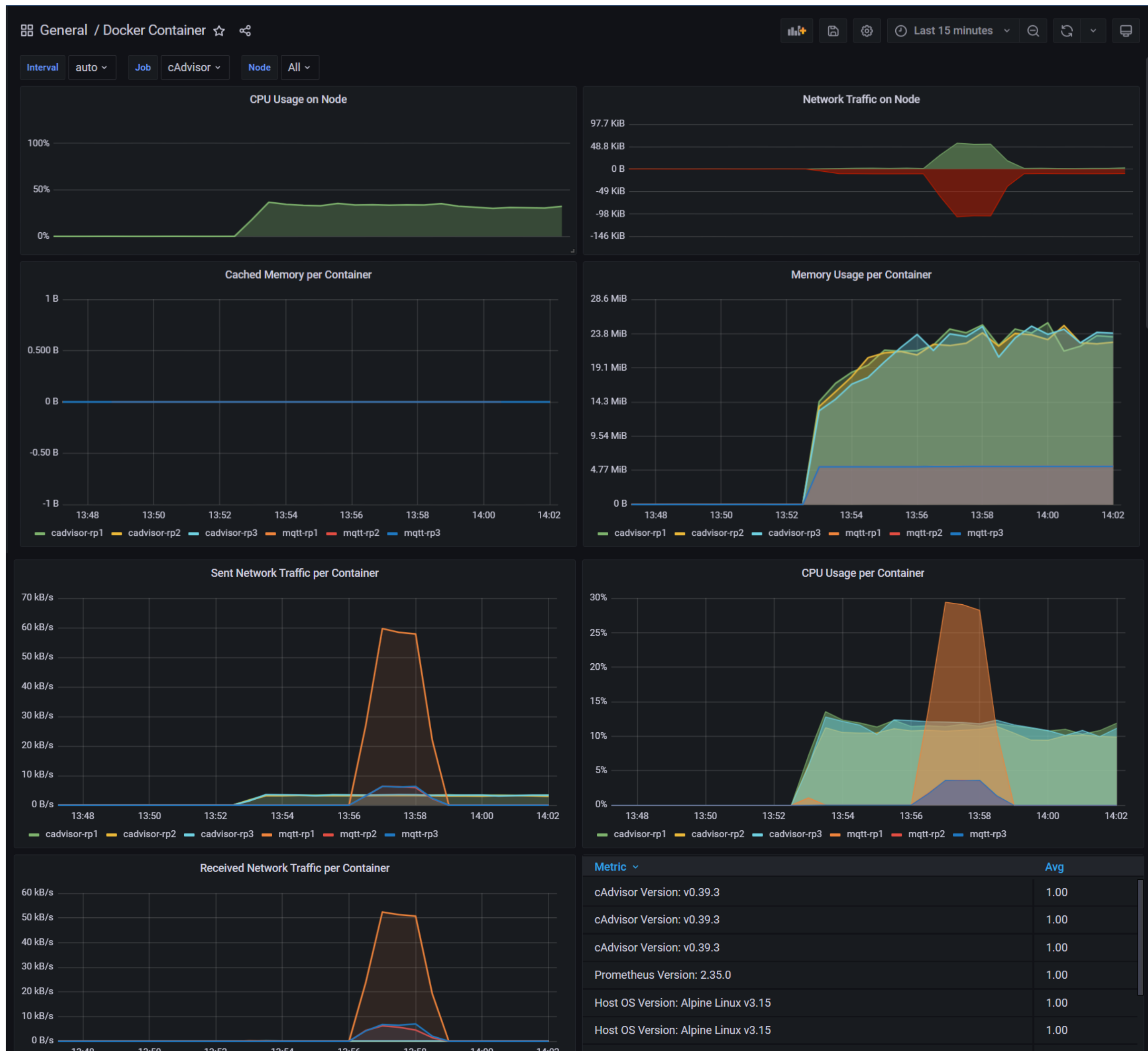


Figure 4: Grafana container metrics for experiment 2

The above image demonstrates the docker container metrics from running the application by using least connections as the load balancing algorithm which is running on the mqtt-rp1 container which then distributes the data to Raspberry Pi two and three. The data sent to Raspberry Pi two and three is processed in mqtt-rp2 and mqtt-rp3 respectively and from here sent to the cloud. We can see in this diagram that all three of the mqtt containers have negligible CPU usage until I send the data from the virtual machine to mqtt-rp1 at 13:56. We can see that mqtt-rp1 uses the most CPU power, and mqtt-rp2 and mqtt-rp3 both have a similar amount of CPU usage. The application eventually crashes due to a “socket timeout error”. The socket timeout error is due to their being too much data for the load balancer to handle.

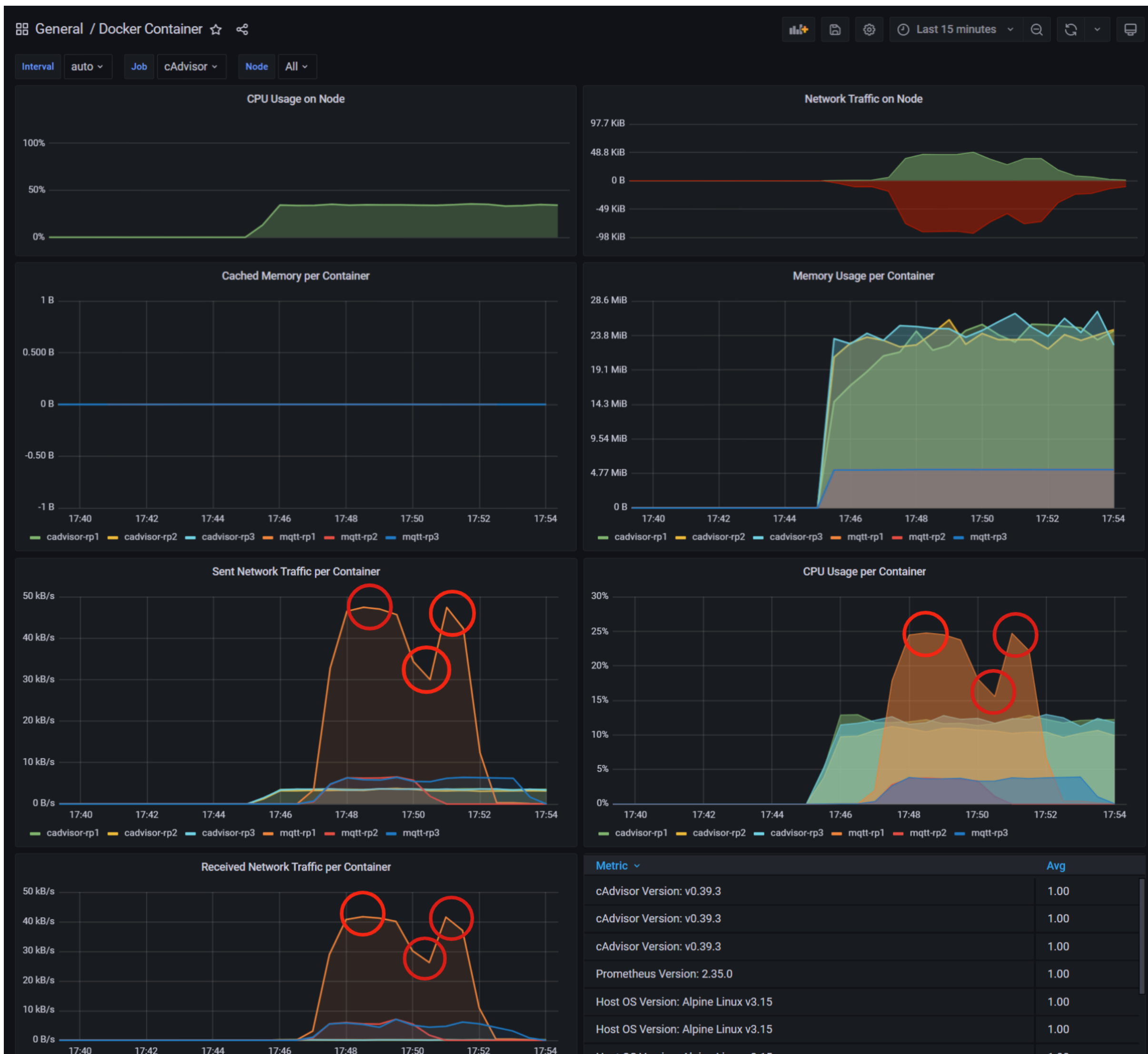


Figure 5: Grafana results for experiment 8 with the trend in peaks and troughs identified

As we can see throughout all of the experiments the CPU usage for the cadvisor containers on the Grafana remains relatively consistent at approximately 12.5% throughout all of the experiments hence this allows us to assume that the CPU usage of the cadvisor containers did not affect the validity of the results. If there was a high variance in the CPU usage of the cAdvisor containers this could affect the amount of CPU power each experiment had access to. We can also see there is a correlation between the sent traffic per container, received traffic per container, and CPU Usage per container. This is demonstrated by the shapes of the graph in experiment 8. As we can see, the peaks and troughs occur at the same point in time and the graphs all follow the same general pattern. This shows us that an increase in received network traffic for a container, causes an increase in CPU usage in the load balancing container, due to more packets needing to be allocated to Pi 2 and Pi 3. This increase in CPU usage means more data is able packets are able to be sent out from the load balancer, hence the increase in Sent Network Traffic per container, also following the same trend. This trend shows us that there is a direct positive link between the amount of messages a container receives and the CPU usage to process this data, hence

demonstrating that by restricting CPU usage of certain docker containers this should impact their ability to process as many messages at the same speed.

We can see that in most experiments the CPU usage of mqtt-rp2 and mqtt-rp3 are around the 5% level, however in experiment 5 mqtt-rp2 only reaches just under 2% (this container was constrained to have 1/16th of the CPU power when compared to mqtt-rp3). This indicates that the CPU constraints implemented on the docker containers are working.

4.7 Evaluating results against the hypothesis

In light of the results the hypothesis has not been proven that “the least connections algorithm will be the most efficient load balancing algorithm when compared with round robin and weighted round robin.” In experiments 1, 2, and 3, experiments 1 and 3 are joint leaders in performance (round robin and weighted round robin). In experiments 4, 6 and 8, as well as experiments 5, 7, and 9, the weighted round robin algorithm is the clear leader here by performance here as well.

Although this data does not prove the hypothesis there are definitely a lot of benefits of the least connections algorithm, when compared with weighted round robin. One benefit of least connections is that this algorithm can be deployed into most systems with very little configuration. On the contrary, with weighted round robin, for it to perform optimally, there has to be an understanding of how powerful all of the edge devices are (assuming they are not equal), and this may be time consuming to implement into a system which has multiple edge devices connected to a load balancer. Furthermore, as the least connections algorithm is dynamic, this means it is more adaptable to inconsistent packet sizes. In the system that was implemented in this project, all of the packets sent over MQTT have been of a similar size and the processing required was relatively consistent, however in a system with varying packet sizes, and where the amount of processing will vary depending on the size of the packet (for instance, if a larger packets contained more values and the average of all the values needed to be calculated), least connections would perform much better in this scenario. This is because if one node becomes overloaded with too much data due to receiving a disproportionate amount of larger packets, then the next few packets would be dynamically allocated to other servers until the number of active connections on the overloaded server has reduced (or all other active connections have increased).

However, the findings do demonstrate, that if the weights of the different nodes in a system are known, and the packet sizes received and distributed by the load balancer are of a consistent size, then weighted round robin is the most efficient option (as there is no need to keep an active communication with all of the nodes that the load balancer is sending data to, and a lot less processing required when determining which node to send the packet to). In a system, when all the nodes have equal processing power, the round robin algorithm is the most efficient as shown by experiment 1, 2 and 3 (or weighted round robin, with each node being assigned a weight of one).

5 Conclusion

5.1 Meeting the objectives

Objective one is “to use the Urban Observatory API (IoT), with the Raspberry Pi acting as an edge device and AWS for the cloud to emulate the data flow of an osmotic computing pipeline”. Objective one has predominantly been a success in this project because a system has been created which emulates the data flow that could happen in a real world osmotic computing system. Evidence of this system was shown in the technical demonstration, and is shown through the code submitted. The data sent from the virtual machine, emulates data that could be sent directly from an IoT sensor or multiple IoT devices (the data read from the Urban Observatory API in the emulation in fact originates from an IoT sensor). The Raspberry Pis emulate the edge layer of an osmotic computing pipeline, as they perform processing on the data and then send the data to the cloud similar to that of a real world application. The data is then also stored in the cloud, therefore most of the requirements of an osmotic computing pipeline have been met because the fundamental system which allows the data to be able to from the IoT device to the edge to the cloud has been implemented. If this project was to be taken further, one could add the functionality so that the application can take a snapshot of container processing data in an edge device and then continue to process this in the cloud or vice versa. I have made this suggestion because the interchangeability of where the data is processed and the ability to migrate data across to a microservice in the cloud from a microservice on the edge or vice versa is key to the concept of osmotic computing. Although the “osmotic” functionality has not been implemented into the system, I do not believe this has affected the ability to investigate the aim or objectives two or three, because fundamentally the methods which have been investigated to mitigate the impact of backpressure, can be applied any Internet of Things Programming model that contains resource constrained edge devices including osmotic computing.

Objective 2 is “to investigate and evaluate the different algorithms that can be used for load balancing”. Objective 2 has been met because the effectiveness of the three load balancing algorithms have been investigated and evaluated by using the experimental results talked about in results and evaluation. Objective three is “to implement an efficient load balancing algorithm into the edge layer of an osmotic computing pipeline.” This objective has also been met because three load balancing algorithms have been implemented in the system. The hypothesis that the least connections algorithm would be the most efficient load balancing algorithm, was not proven but weighted round robin was the most efficient algorithm in all of the experiment comparisons involving the three different algorithms. As a result of this we can conclude that weighted round robin is the most efficient algorithm for the configurations that were tested. As this algorithm was implemented into the edge layer of an application that mostly emulates an osmotic computing pipeline, it is reasonable to conclude that objective 3 has been met.

Objectives 4 and 5 from original proposal were omitted because the load balancing algorithm used can often be the bottleneck in the system which causes more back-pressure to build up and causes the system to break more easily than if a more efficient load balancing algorithm had been used. Using methods of adaptive data flow control methods (such as the token

bucket algorithm, or limiting the data forwarding rate, or data receiving rate) only mitigate the effect that a low efficiency load balancing algorithm has. For example by limiting the receiving rate in the load balancer, this may ensure that Raspberry Pi 2 and Raspberry Pi 3 do not crash, however fundamentally if the load balancing algorithm used is more efficient, this may not be necessary and if the receiving rate is limited this will mean that packets could be lost if the data is being transmitted with a QoS of 0 or if it is being transmitted with a QoS of 1, then the data will be received with a delay.

5.2 Justification for omission of objectives 4 and 5

Omitting objectives 4 and 5 helped to achieve objectives 2 and 3 because this meant that more time could be spent running different experiments and gave the project a narrower scope so that more emphasis could be placed on comparing the 3 load balancing algorithms.

The overall aim of the project is "to evaluate the effectiveness of different methods to deal with backpressure in an osmotic computing pipeline." I believe this has been met as a result of meeting objectives 1, 2, and 3.

As mentioned in the results and evaluation, the experiments provide useful insights into the proposed hypothesis, but they do not prove that the hypothesis is correct. In spite of this, a lot has been achieved from this project and the implementation used in this project could be used to provide a strong foundation for future work.

5.3 Suggestions for Future Work

Here are some suggestions for further work, for those who want to take this project further:

- Implement data flow control methods such as the token bucket algorithm, leaky bucket algorithm, or limiting the forwarding rate of the load balancer rate in conjunction with the load balancing algorithms used in this project and compare the metrics to those of the experiments already run.
- Run the experiments several times and take an average of these values so the results are more reliable and anomalies can be better identified.
- Use a larger data set (or loop through the same data multiple times), to ensure part of the application always eventually crashes (usually via a socket timeout). This time taken until the application breaks could also be recorded and used to evaluate the effectiveness of the different algorithms at dealing with back-pressure (assuming the data is consistently sent from the virtual machine at the same rate).
- Perform the experiments using other known existing load balancing algorithms to evaluate their effectiveness.
- Create a new load balancing algorithm to compare with the already existing algorithms such as those used in this project.

References

Buzachis, A. *et al.* (2021) 'Modeling and Emulation of an Osmotic Computing Ecosystem using OsmoticToolkit', in *2021 Australasian Computer Science Week Multiconference*. New York, NY, USA: Association for Computing Machinery (ACSW '21, 9), pp. 1–9.

cadvisor: Analyzes resource usage and performance characteristics of running containers (no date). Github. Available at: https://github.com/google/cadvisor (Accessed: 25 May 2022).

Configure Grafana Docker image (no date) *Grafana Labs*. Available at: https://grafana.com/docs/grafana/latest/administration/configure-docker/ (Accessed: 26 May 2022).

Docker Container (no date) *Grafana Labs*. Available at: https://grafana.com/grafana/dashboards/11600 (Accessed: 25 May 2022).

Docker Documentation (2022) *Docker Documentation*. Available at: https://docs.docker.com/ (Accessed: 27 May 2022).

How to Connect to your EC2 Instance using PuTTY V1.1 (no date). Available at: https://asf.alaska.edu/how-to/data-recipes/connect-to-your-ec2-instance-using-putty-v1-1/ (Accessed: 25 May 2022).

Knoldus Inc. (2022) *Getting started with Grafana and Prometheus*, *Medium*. Available at: https://medium.com/@knoldus/getting-started-with-grafana-and-prometheus-4176c1408396 (Accessed: 25 May 2022).

Mothish (2021) *Containerizing a Python app — MySQL + Python + Docker*, *Medium*. Available at: https://mothishdeenadayalan.medium.com/containerizing-a-python-app-mysql-python-docker-1ce64e444ed9 (Accessed: 27 May 2022).

Nardelli, M. *et al.* (2017) 'Osmotic Flow: Osmotic Computing + IoT Workflow', *IEEE Cloud Computing*, 4(2), pp. 68–75.

Phelps, J. (2019) *Backpressure explained — the resisted flow of data through software*, *Medium*. Available at: https://medium.com/@jayphelps/backpressure-explained-the-flow-of-data-through-software-2350b3e77ce7 (Accessed: 4 March 2022).

Rausch, T., Dustdar, S. and Ranjan, R. (2018) 'Osmotic Message-Oriented Middleware for the Internet of Things', *IEEE Cloud Computing*, 5(2), pp. 17–25.

Sharma, S., Singh, S. and Sharma, M. (2008) 'Performance analysis of load balancing algorithms', *Proceedings of World Academy of Science, Engineering and Technology*, 38(3), pp. 269–272.

Villari, M. *et al.* (2016) 'Osmotic Computing: A New Paradigm for Edge/Cloud Integration', *IEEE Cloud Computing*, 3(6), pp. 76–83.

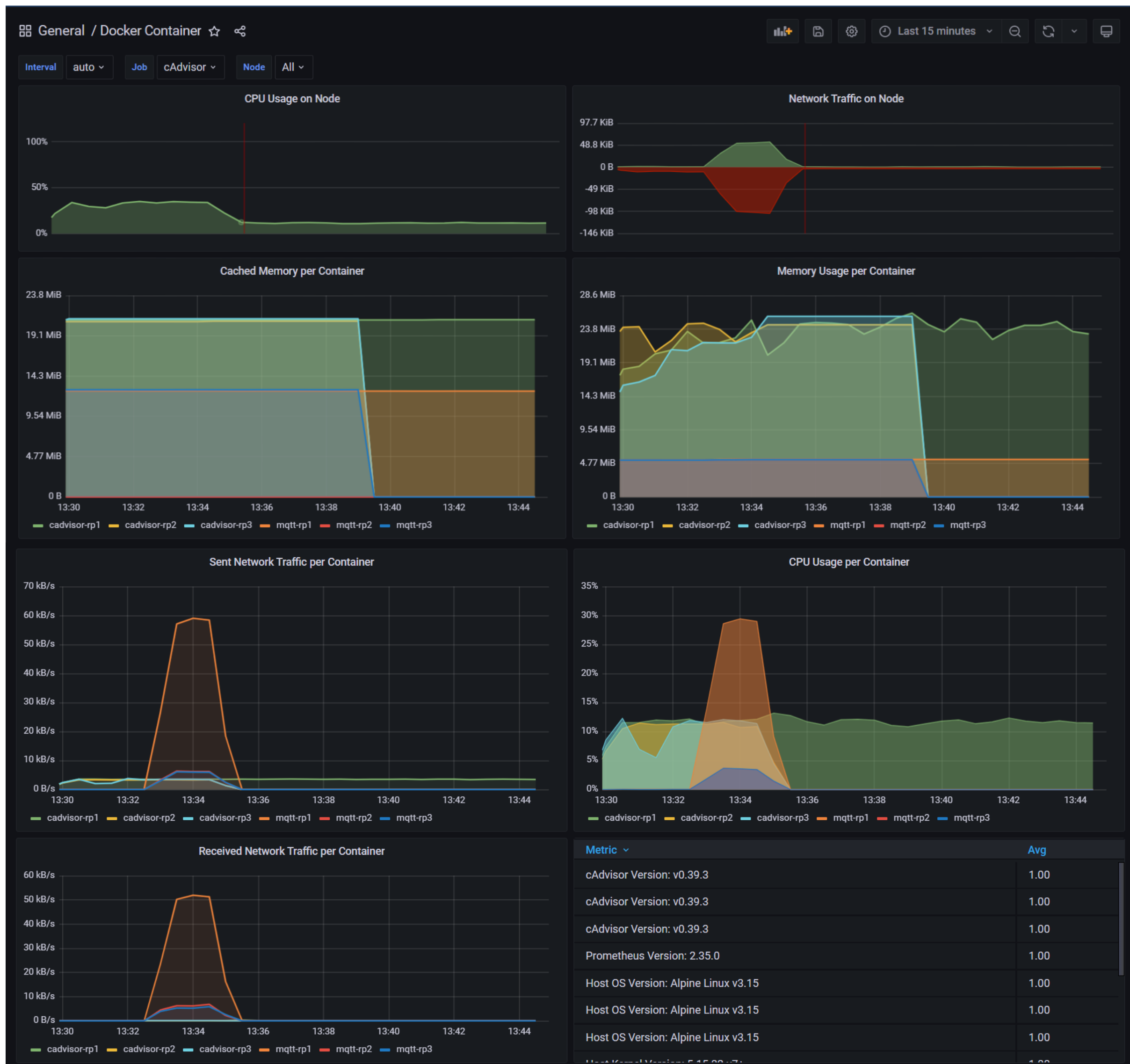
What is Prometheus and Why Should You Use It? **(2020)** ***Opsani*****. Available at: https://opsani.com/resources/what-is-prometheus-and-why-should-you-use-it/ (Accessed: 25 May 2022).**

Which cgroup flags to add to cmdline.txt on 64bit Bullseye for Docker/Kubernetes - Raspberry Pi Forums **(no date). Available at: https://forums.raspberrypi.com/viewtopic.php?t=325521 (Accessed: 26 May 2022).**

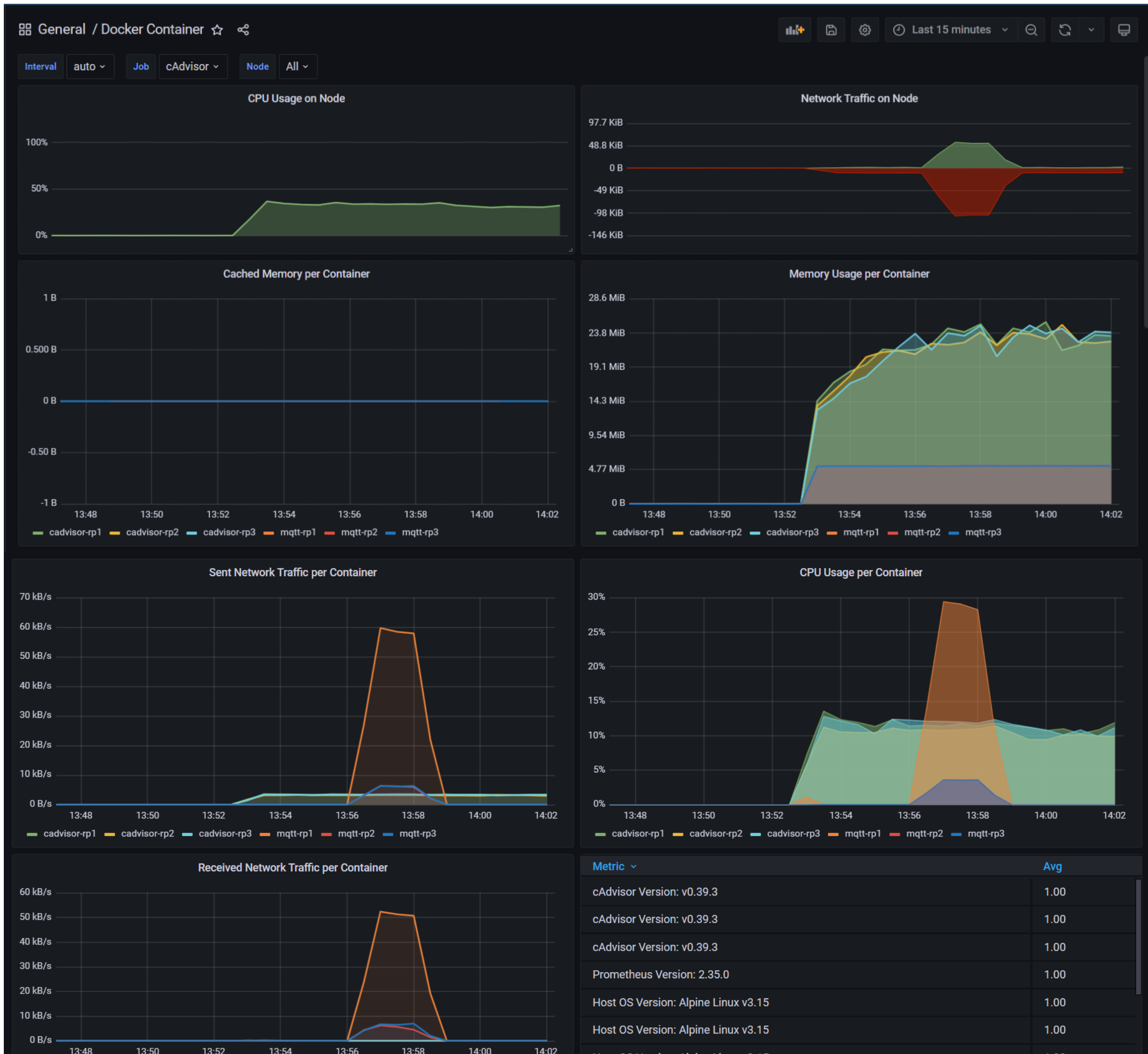
ZCube (no date) ***cadvisor-docker: cAdvisor container*****. Github. Available at: https://github.com/ZCube/cadvisor-docker (Accessed: 25 May 2022).**

Appendix

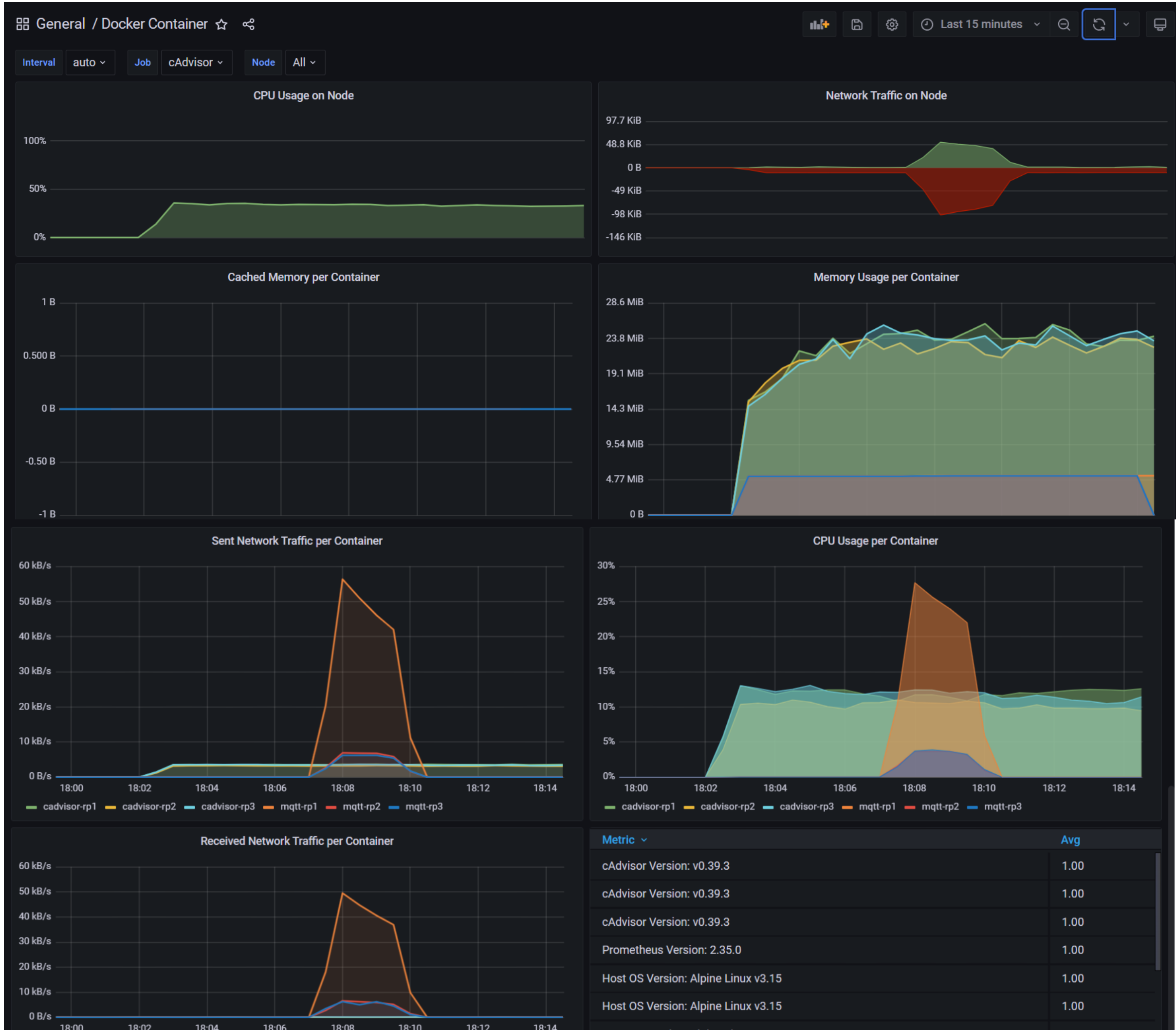
Grafana Graph Experiments Results



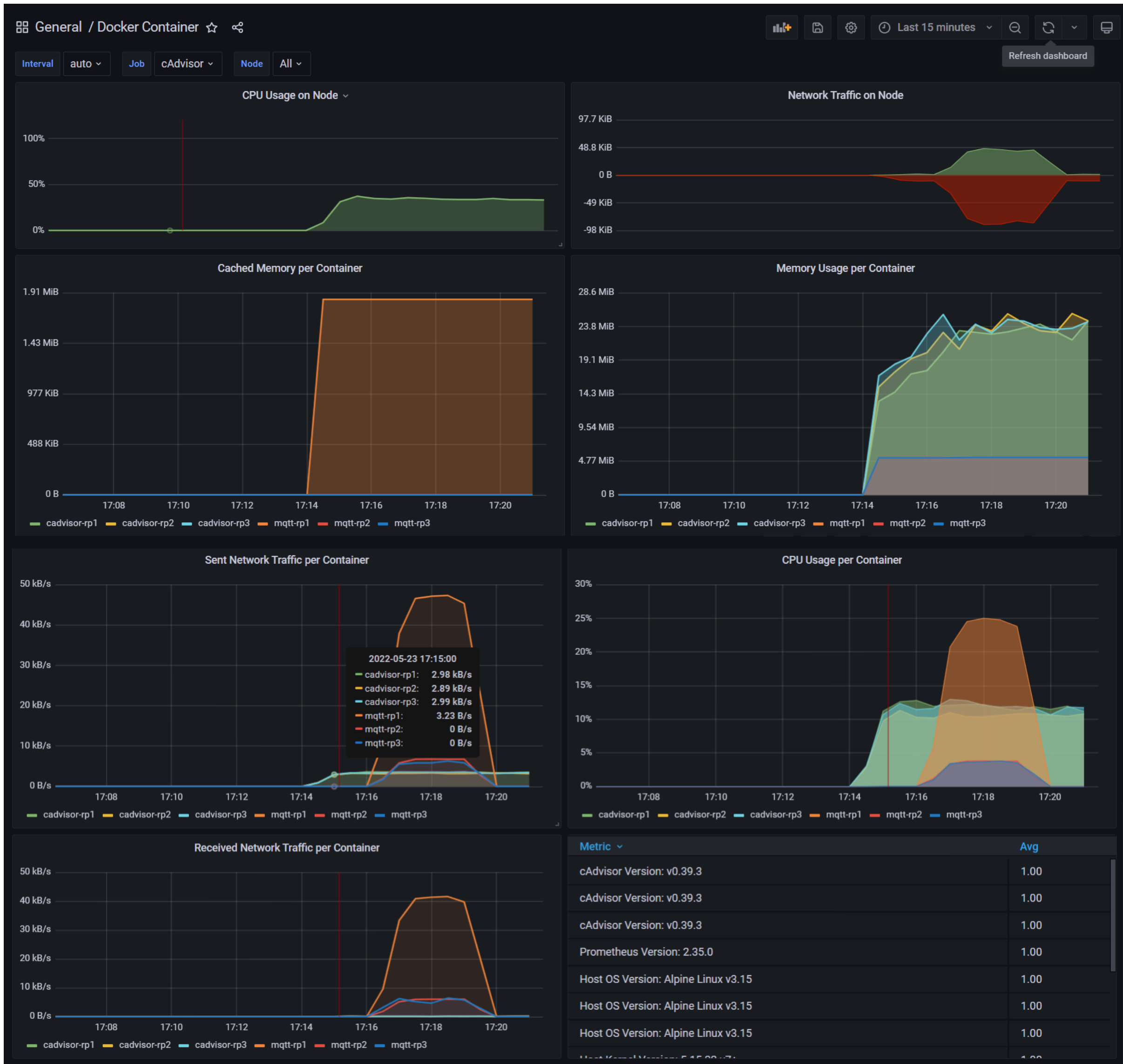
Experiment 1



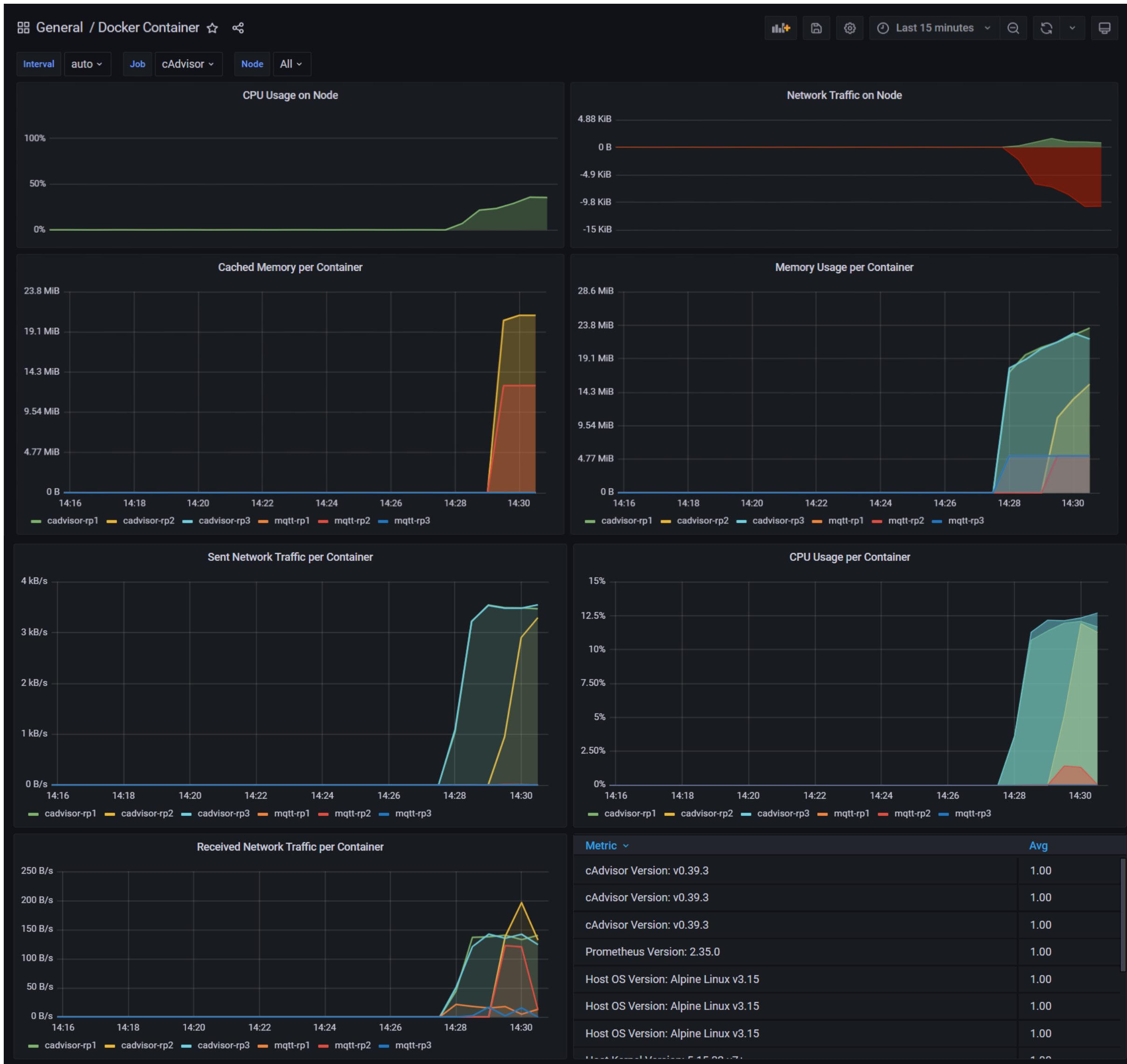
Experiment 2



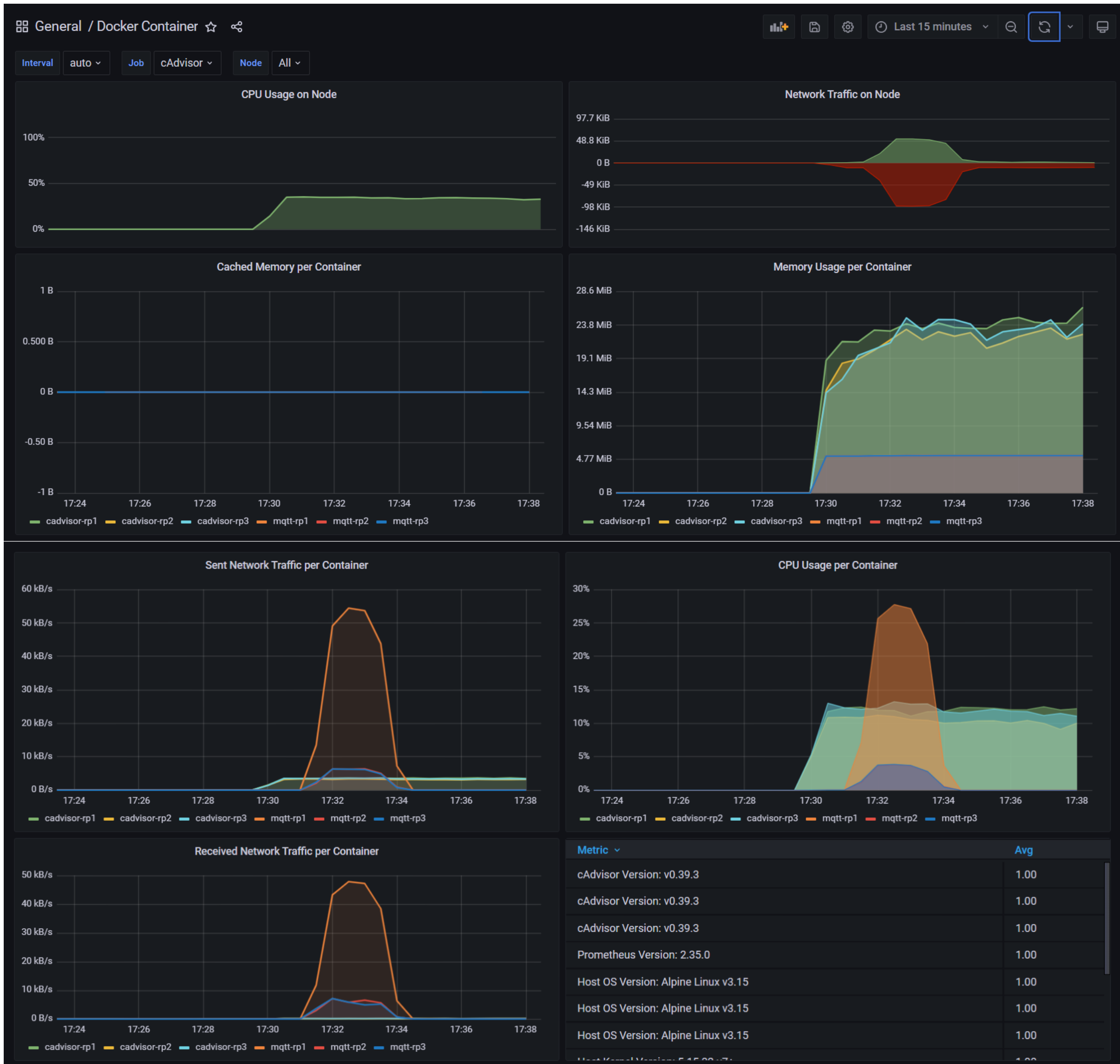
Experiment 3



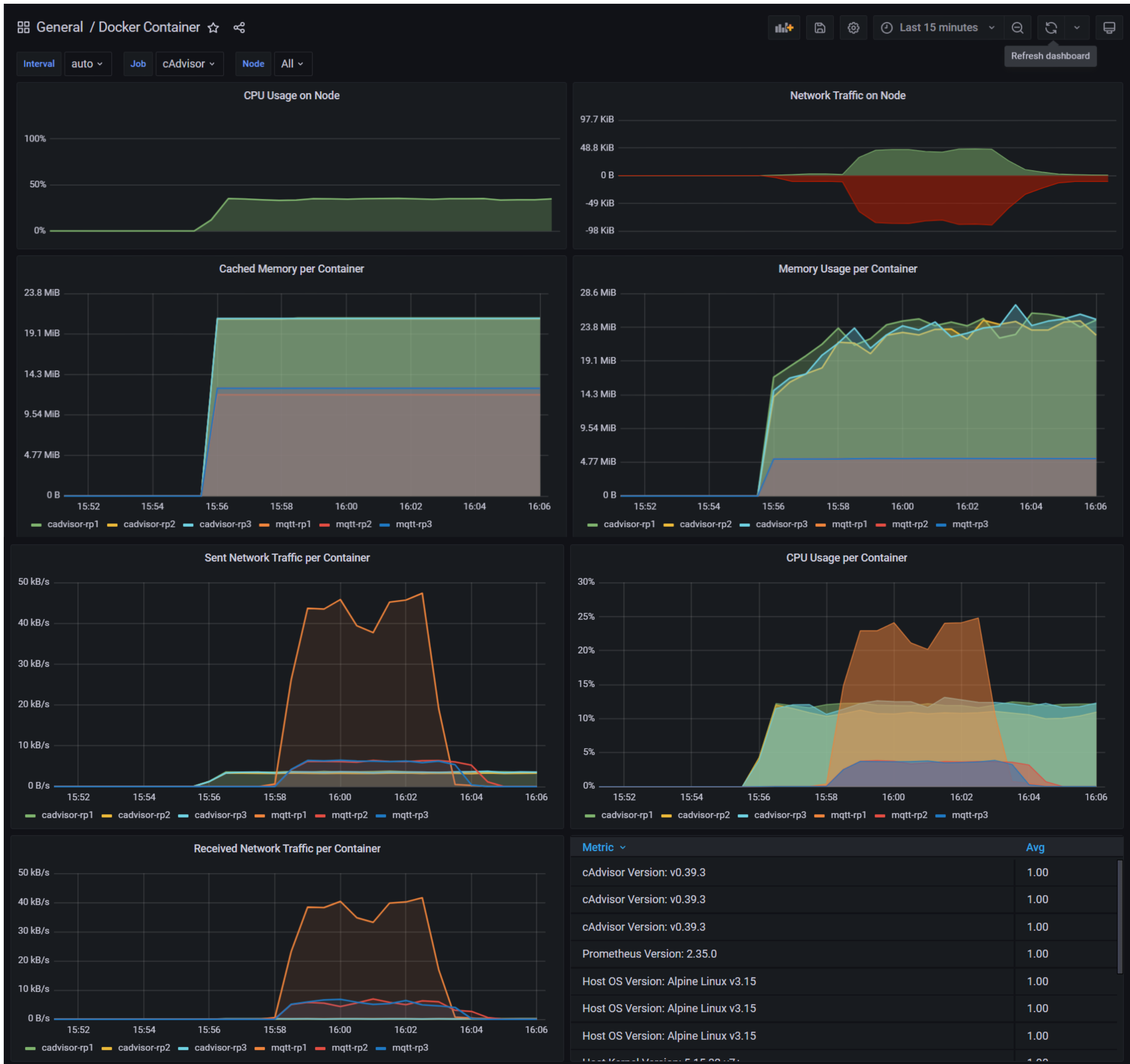
Experiment 4



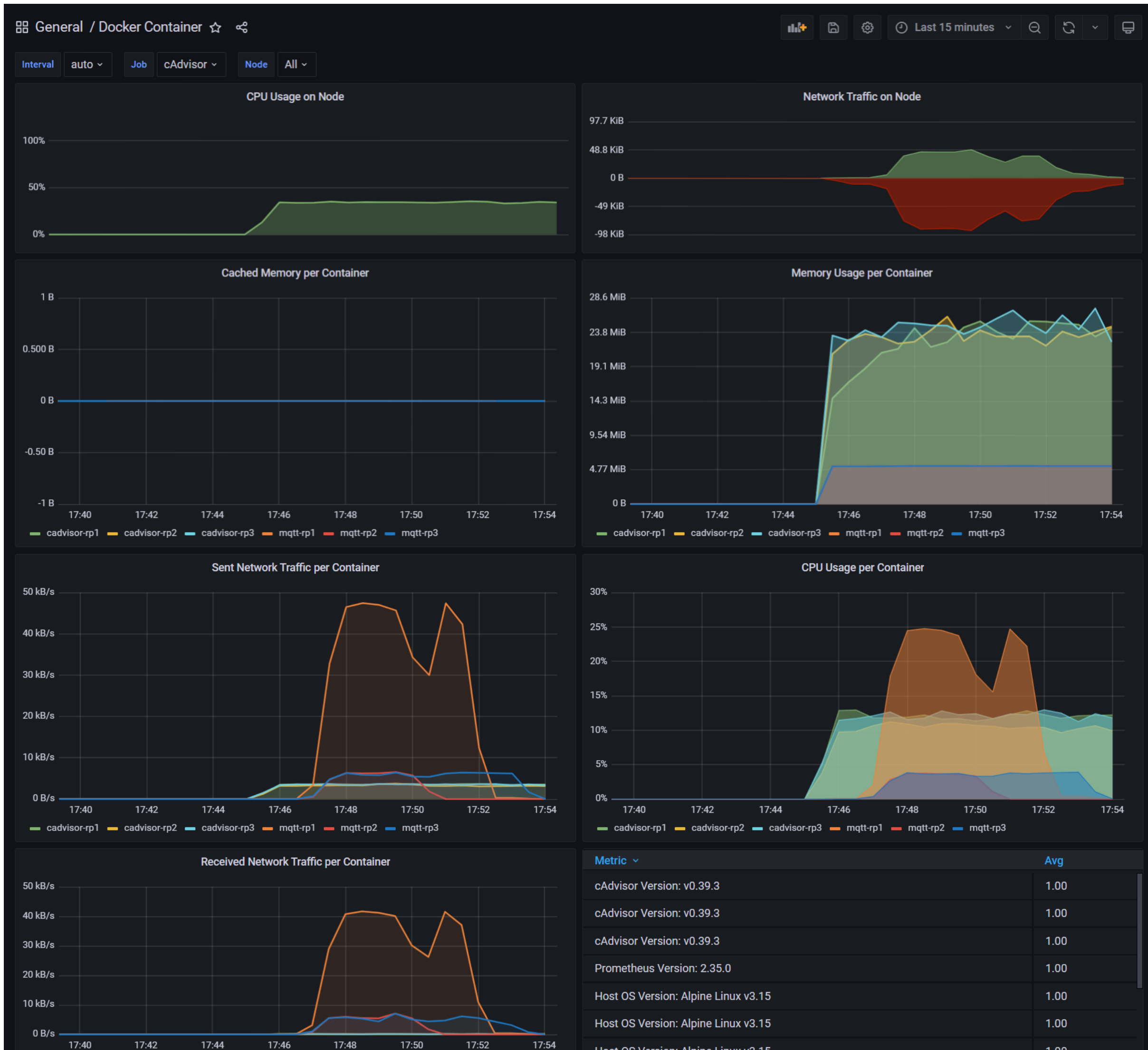
Experiment 5



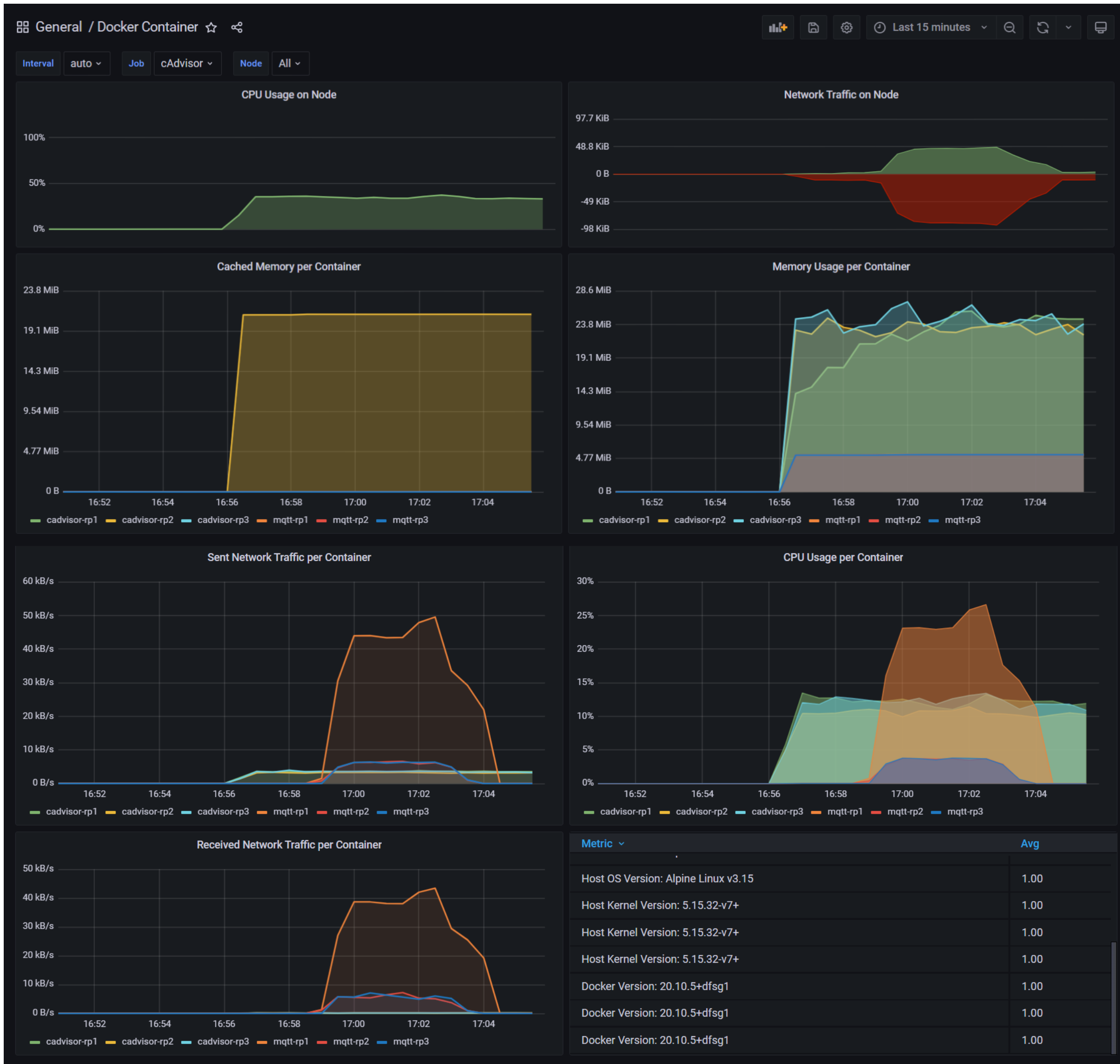
Experiment 6



Experiment 7



Experiment 8



Experiment 9

SQL Database Results

Experiments:

1. Round Robin - CPU constraints: none
2. Least Connections - CPU constraints: none
3. Weighted Round Robin - weights: 1:1 - CPU constraints: none
4. Weighted Round Robin - weights: 1:4 - CPU constraints 1:4
5. Weighted Round Robin - 1:16 CPU 0.25:4 CPU constraints: 0.25:4
6. Round Robin - CPU constraints: 1:4
7. Round Robin - CPU constraints: 0.25:4
8. Least Connections - CPU constraints: 1:4
9. Least Connections - CPU constraints: 0.25:4

Averages of all Data - unrounded

Experiment Number	Pi to cloud (ms)	Load balancer to cloud (ms)	VM to cloud (ms)
Experiment 1	260.3655663522527	30958.31457103934	30978.432379204995
Experiment 2	253.73382739415246	37428.812415516564	37446.14644892253
Experiment 3	256.50002318113195	34327.78953824815	34340.34880420003
Experiment 4	257.26317298326785	40576.89944302732	40592.89080754807
Experiment 5	258.29354971548565	30653.64272101274	30672.89371191125
Experiment 6	254.15345624467017	35866.766690966106	35884.013298545084
Experiment 7	262.4518091473545	46410.67729377371	46426.05521884249
Experiment 8	251.29886727518848	46946.53382454909	46965.014011420215
Experiment 9	251.84729930486526	41846.82542295985	41863.53593965891

Averages of Data - top 1,000 results - unrounded

Experiment Number	Pi to cloud (ms)	Load balancer to cloud (ms)	VM to cloud (ms)
Experiment 1	260.37013774108885	31011.60660494995	31031.772082855223
Experiment 2	253.719279006958	37229.497666793824	37246.83436856079
Experiment 3	256.4359038543701	30955.817676879884	30967.973732055663
Experiment 4	257.63647900390623	38297.032420684816	38313.64253173828
Experiment 5	257.22581932067874	28563.602897598266	28582.772873687743
Experiment 6	254.2340662689209	39088.312412353516	39105.89640817261
Experiment 7	260.8330633087158	35085.58697161865	35102.135136993405
Experiment 8	250.62706884765626	44513.04099786377	44529.122443237306
Experiment 9	249.7225714111328	34950.44953280639	34964.834082122805

Averages of All Data - 3 s.f

Experiment Number	Pi to cloud (ms)	Load balancer to cloud (ms)	VM to cloud (ms)
Experiment 1	260	31000	31000
Experiment 2	254	37400	37400
Experiment 3	257	34300	34300
Experiment 4	257	40600	40600
Experiment 5	258	30700	30700
Experiment 6	254	35900	35900
Experiment 7	262	46400	46400
Experiment 8	251	46900	47000
Experiment 9	252	41800	41900

Averages of Data - top 1,000 rows - 3 s.f

Experiment Number	Pi to cloud (ms)	Load balancer to cloud (ms)	VM to cloud (ms)
Experiment 1	260	31000	31000
Experiment 2	254	37200	37200
Experiment 3	256	31000	31000
Experiment 4	258	38300	38300
Experiment 5	257	28600	28600
Experiment 6	254	39100	39100
Experiment 7	261	35100	35100
Experiment 8	251	44500	44500
Experiment 9	250	35000	35000