FORECASTING RESOURCE AVAILABILITY FOR SUPPORTING REAL-TIME CONTAINER MIGRATION

BY

VISHAKH SURESH BABU

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

Adviser:

Research Assistant Professor Sibin Mohan

ABSTRACT

Hard real-time applications have stringent response time requirements and failure to meet the deadlines can have catastrophic consequences. With the success of the edge computing paradigm, more edge applications have real-time requirements. User mobility and multi-tenancy on the edge nodes create new challenges for these real-time applications running inside containers. These problems can be resolved by migrating the container along with all of the underlying files to a new node where the application can run safely. However, single-target migration is prone to failure due to network issues or system crashes at the target. It is also hard to guarantee that applications will meet their real-time requirements during (or even after) the migration process.

To this end, we propose a framework for planning reliable real-time container migration. This framework monitors the container and nodes in the network and accurately predicts the future resource availability of the nodes. The forecasts are used to identify a subset of nodes that have sufficient resources to run the container. We migrate to multiple nodes in parallel to make it resilient to network issues and crashes.

ACKNOWLEDGMENTS

First and foremost, I would like to express my sincere gratitude to my adviser Dr. Sibin Mohan for giving me an opportunity to work on this project. I am very grateful for his feedback, suggestions and guidance over the course of this project.

I would also like to thank Bin-Chou Kao and Devikrishna Radhakrishnan for the discussions and input on this project.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Real-time systems refer to systems that respond to an external input within a finite time limit. The correctness of these systems depend not only on the accuracy of the results, but also on whether they meet the timing guarantees, often referred to as 'deadlines'. There are two types of real-time systems: soft and hard. Soft real-time systems may miss a deadline. The consequences of missing the deadline are often not severe. However, in hard real-time systems, failure to meet the timing guarantees has extreme consequences. Missing a deadline might render the whole system useless. For example, consider a system for deploying a vehicle's airbags in the event of a crash. If the system fails to deploy the airbag on time, the safety of the human occupant is compromised. Many cyber-physical systems used in the automobile industry, aviation, healthcare, manufacturing, robotics have such stringent timing requirements.

The success of the edge computing paradigm has brought about an increased use of edge applications with real-time requirements. This creates a new set of challenges [1]. In scenarios where the user of the application is mobile, the response times might go up as the user moves away from the edge node where the computation is hosted. This is not good for latency-critical applications. We can improve the Quality of Service (QoS) by moving the computation to another edge node that is closer to the user. Another key challenge in edge computing is resource management [2]. The physical resources on an edge node are often shared among multiple tenants. If one of the tenants uses an excessive amount of resources, it might affect the performance of the other applications running on that node. The desired QoS can be delivered by moving the application to another node that has enough resources.

The applications run in virtualized environments on the edge. Container technology (OS-level virtualization) has become a very popular lightweight alternative to virtual machines due to their ease of deployment and superior performance [3]. For containers running on the edge nodes, live migration can be performed to resolve the aforementioned problems arising from user mobility or multi-tenancy. The application is migrated with all of its runtime state and restored at the destination.

Migration over the network comes with an element of risk. A migration attempt might fail for a variety of reasons - for example, the target node might crash while the migration is taking place. If an attempt fails, we could restart and try migrating to another node in the network. Such an approach might work well for non real-time applications without any deadlines. However, it is not suited for safety or mission critical applications that often have response time requirements in milliseconds. It is also hard to guarantee that applications

will meet their real-time requirements while running at the destination node.

We can perform the live migration in a more fault tolerant manner by attempting migrations to multiple destinations in parallel. In order to do this, we must identify a set of nodes that have sufficient resources to run the container. In this work, we show how this can be done by tracking and forecasting the resources available on the nodes in the network.

Research Goal. This thesis makes the following hypothesis:

> *it is possible to forecast the future resource availability of remote nodes by monitoring their system states and resource usage; the forecasts can enable additional capabilities such as fault tolerant migration, load balancing, etc.*

There are three questions that this thesis serves to answer:

1. How do we profile containers and remote hosts to track their resource usage?
2. How can we project the resource usage to model the system state after migration?
3. How can we use the forecasts to identify nodes that have enough resources to run the real-time container?

Contributions. The main contributions of this work can be summarized as:

- Development of a monitoring framework that can track the container and nodes.
- Design of an analysis engine that uses the information collected to forecast the resource availability and plan a reliable migration of real-time containers.
- Implementation and evaluation of our proposed framework.

Outline. The rest of this thesis is organized as follows. Chapter 2 provides some background information related to this work. In chapter 3, we describe the overall design of our framework and list the functions of each component in our system. Chapters 4-6 explain each component of our framework in detail. In chapter 7, we describe the experiments we ran and present our results. Chapter 8 talks about other research related to this work. In chapter 9, we discuss the limitations of our work and possible future directions.

CHAPTER 2: BACKGROUND

In this chapter, we provide some background information on virtualization and two statistical models we have used for fitting the data (Chapter 5).

2.1 VIRTUALIZATION

Virtualization has been widely used in the industry. It provides a number of benefits ranging from reduced upfront and operating costs to dynamic scaling of allocated resources. There are two main virtualization technologies that have become popular over the years - virtual machines (VMs) and containers. Figure 2.1 shows how these two technologies differ in their architecture.

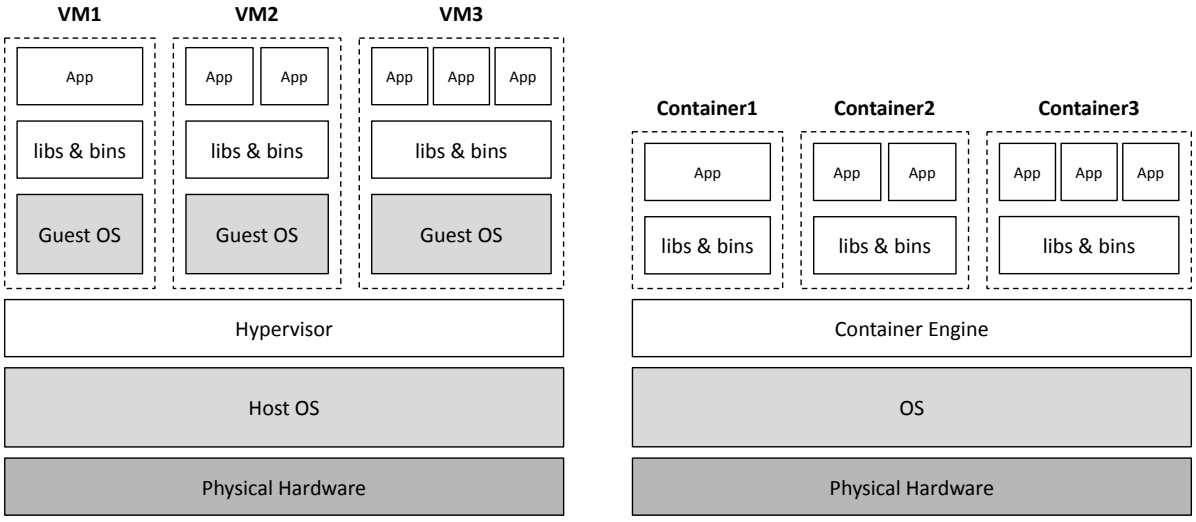

Figure 2.1: Virtual Machines (left) vs. Containers (right)

Virtual machines abstract out the underlying hardware and allow them to be shared among multiple virtual instances of the system. Each virtual machine can run its own copy of an operating system. A hypervisor (virtual machine monitor) sits between the host operating system and virtual machines, and manages resource allocation, communication, etc.

Unlike virtual machines, containers virtualize only the host operating system (OS-level virtualization). They are much faster [3, 4] and portable as they don't have a guest operating system in every container instance. Containers provide a light-weight way to package an application along with its supporting files.

Containers have grown in popularity since the introduction of Docker in 2013. Docker [5] is an open-source platform that allows us to build, test and deploy containerized applications. Podman [6] is another tool for building containers. It is compatible with Docker and can launch containers using Docker container images. Unlike docker, podman is daemonless; it launches containers as child processes. The docker dameon is a single point of failure in the system. As a result, podman containers are inherently more secure. We use podman containers in this work.

2.2 ARIMA MODEL

The AutoRegressive Integrated Moving Average (ARIMA) model [7] is a statistical model that is used for forecasting univariate time series data. An ARIMA model is a combination of two simpler models:

- An *AutoRegressive (AR) model* is one in which the value of a variable is estimated using a linear combination of its previous (lagged) values. In an AR model of order p (written AR(p)), the value of a variable x at time t can be expressed as:

$$x_t = c + \phi_1 x_{t-1} + \phi_2 x_{t-2} + ... + \phi_p x_{t-p} + \epsilon_t \tag{2.1}$$

  where $x_{t-1}, x_{t-2}, ..., x_{t-p}$ are p lagged values of x; $\phi_1, \phi_2, ..., \phi_p$ are parameters of the AR model; c is a constant and ϵ_t is noise.

- A *Moving Average (MA) model* uses the past prediction errors to forecast a variable. In a MA model of order q (written MA(q)), the current value can be expressed as a linear combination of q previous error terms as:

$$x_t = \mu + \epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + ... + \theta_q \epsilon_{t-q} \tag{2.2}$$

  where $\epsilon_t, \epsilon_{t-1}, \epsilon_{t-2}, ..., \epsilon_{t-q}$ are prediction errors; $\theta_1, \theta_2, ..., \theta_q$ are parameters of the MA model; μ is the mean of the time series.

An ARMA(p,q) model [8] is a combination of an AR(p) and MA(q) model and can be represented mathematically as:

$$x_t = c + \sum_{i=1}^{p} \phi_i x_{t-i} + \epsilon_t + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} \tag{2.3}$$

The observations are differenced to produce a stationary time series. A time series is said to be stationary if its properties are stable over time i.e., they remain same regardless of when series is observed. In other words, the time series is devoid of any seasonality. The number of times the series is differenced is the order of differencing (d) of the ARIMA(p,d,q) model.

2.3 LEAST SQUARES FITTING

Consider n data samples x_1, x_2, x_3, x_4, ..., x_n collected at times t_1 through t_n. We would like to find a polynomial of degree k (with $k < n$), that best fits this series of data points. A degree k polynomial has the form:

$$x = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + ... + a_k t^k \tag{2.4}$$

where a_0, a_1, ..., a_k are constants. The residual (difference between the actual and estimated value) for a data point at time t_i is $r_i = x_i - \hat{x_i} = x_i - (a_0 + a_1 t_i + ... + a_k t_i^k)$. The values of the coefficients for the best fit curve are obtained by minimizing the sum of squares of the residuals [9], computed as:

$$\mathcal{E} = \sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} \left\{ x_i - (a_0 + a_1 t_i + ... + a_k t_i^k) \right\}^2 \tag{2.5}$$

To find the values of the coefficients at which $\mathcal{E}$ attains it minimum value, we equate the partial derivatives of $\mathcal{E}$ with respect to each a_j to 0 [10], i.e.,

$$\frac{\partial \mathcal{E}}{\partial a_j} = 0, \quad \forall j \in \{0, 1, 2, ..., k\} \tag{2.6}$$

$$2 \sum_{i=1}^{n} \left\{ x_i - (a_0 + a_1 t_i + ... + a_k t_i^k) \right\} \cdot (-t_i^j) = 0, \quad \forall j \tag{2.7}$$

This produces a system of $k+1$ equations that can be expressed in matrix form as follows:

$$\begin{bmatrix} \sum_{i=1}^n t_i^0 & \sum_{i=1}^n t_i^1 & \sum_{i=1}^n t_i^2 & \dots & \sum_{i=1}^n t_i^k \\ \sum_{i=1}^n t_i^1 & \sum_{i=1}^n t_i^2 & \sum_{i=1}^n t_i^3 & \dots & \sum_{i=1}^n t_i^{k+1} \\ \sum_{i=1}^n t_i^2 & \sum_{i=1}^n t_i^3 & \sum_{i=1}^n t_i^4 & \dots & \sum_{i=1}^n t_i^{k+2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^n t_i^k & \sum_{i=1}^n t_i^{k+1} & \sum_{i=1}^n t_i^{k+2} & \dots & \sum_{i=1}^n t_i^{2k} \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_k \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^n x_i \\ \sum_{i=1}^n x_i t_i \\ \sum_{i=1}^n x_i t_i^2 \\ \vdots \\ \sum_{i=1}^n x_i t_i^k \end{bmatrix} \tag{2.8}$$

The augmented matrix for this system is,

$$M = \begin{bmatrix} n & \sum_{i=1}^{n} t_i & \sum_{i=1}^{n} t_i^2 & \dots & \sum_{i=1}^{n} t_i^k & \sum_{i=1}^{n} x_i \\ \sum_{i=1}^{n} t_i & \sum_{i=1}^{n} t_i^2 & \sum_{i=1}^{n} t_i^3 & \dots & \sum_{i=1}^{n} t_i^{k+1} & \sum_{i=1}^{n} x_i t_i \\ \sum_{i=1}^{n} t_i^2 & \sum_{i=1}^{n} t_i^3 & \sum_{i=1}^{n} t_i^4 & \dots & \sum_{i=1}^{n} t_i^{k+2} & \sum_{i=1}^{n} x_i t_i^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \sum_{i=1}^{n} t_i^k & \sum_{i=1}^{n} t_i^{k+1} & \sum_{i=1}^{n} t_i^{k+2} & \dots & \sum_{i=1}^{n} t_i^{2k} & \sum_{i=1}^{n} x_i t_i^k \end{bmatrix} \tag{2.9}$$

The augmented matrix M can be transformed to its reduced row echelon form using Gauss-Jordan elimination [11, 12]. For every matrix, the reduced row echelon form is unique and does not depend on how the row reduction is performed. Once we compute the reduced form, the last column gives the values of the coefficients a_0 through a_k.

CHAPTER 3: SYSTEM DESIGN

The system consists of a set of heterogeneous nodes with different resource capacities. We assume a closed system where the IP address of each node in the system is known in advance. New nodes cannot enter the system at any point in time. However, existing nodes can go offline (due to a crash or network failure) and come back online later. Nodes can communicate and exchange information with any other node in the system.

The application with timing requirements executes a container that is running on a node. The container consumes resources such as CPU and memory for executing the application. When we migrate the container to a different node in the network, it is important to ensure that the destination node has sufficient resources to allow the container to run smoothly *while meeting the real-time guarantees of the application* running in it.

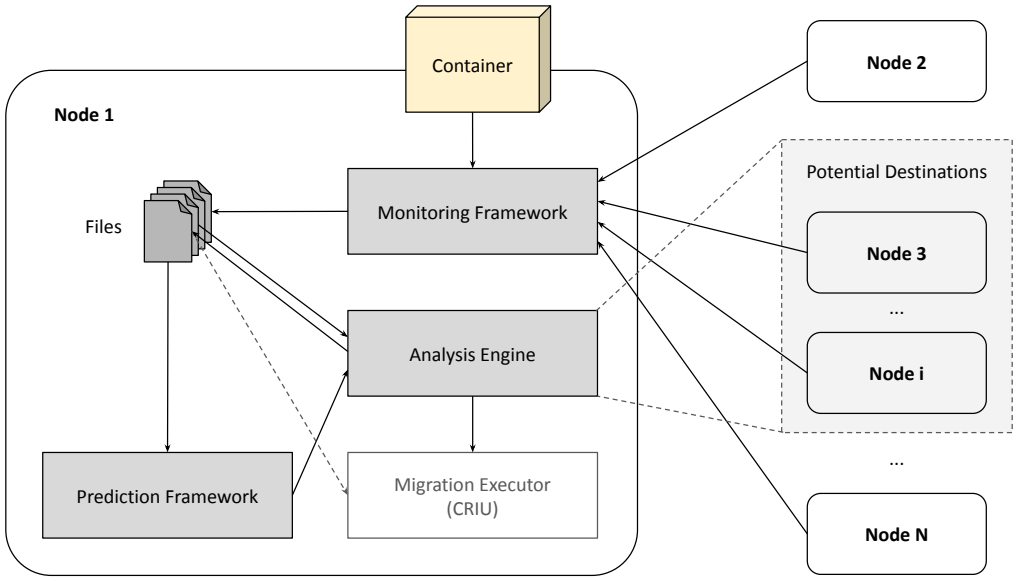

Figure 3.1: Overall architecture of our framework

Figure 3.1 shows the architecture of our framework for migration. Here, the system has N nodes and the podman container [6] (to be migrated) is running on node 1. There are four main components that work together to migrate the container:

- ***Monitoring Framework.*** Records the resource usage statistics of the container and resource availability of the node. It also collects information regarding the resource availability of the other nodes. In addition to this, the monitoring framework keeps

track of the status of the network to identify any targets that might be offline and unsuitable for migration.

- ***Prediction Framework.*** Uses the information collected by the monitor to predict the resource availability at the nodes.

- ***Analysis Engine.*** Analyzes the data recorded for node 1, to figure out if and when there is a need for migrating the container. It also uses the forecasts of the resource availability on the other nodes to determine a subset of nodes that are suitable targets for migration.

- ***Migration Executor.*** Handles the live migration process. It uses CRIU [13] pre-dump to dump the container's memory pages, which are then copied to the destination. The migration executor checkpoints the application at the source node and syncs the container file systems at the source and destination. The container is later restored (from the checkpoint) at the destination.

Multiple destinations are selected from the ranked list produced by the analysis engine. The migration executor initiates migration to all these destinations in parallel. When any attempt succeeds, the migration executor is notified and it kills the other migrations.

The monitoring framework, prediction framework and the analysis engine are described in chapters 4 through 6. *The migration executor is not part of this thesis.*

CHAPTER 4: MONITORING FRAMEWORK

The monitoring framework allows us to profile the containers and nodes in the network. Proper monitoring of the available resources is important as it allows us to identify any machine that is overloaded and offload some of the computation to other machines not at their maximum capacity. Resource monitoring has been a topic of interest in the industry and research for quite some time. Some of the available monitoring solutions collect information from API endpoints [14, 15], while others extract the data from the root file system [16].

When we monitor the system for an extended period of time, the large amounts of data captured can cause the logs to grow very fast. Storage space becomes an obvious concern when we have to log the resource usages for all hosts and running containers for long periods of time. So, we use a custom framework that monitors the host and containers running in the network and log only the information needed for planning the migration. The framework collects CPU and memory data to keep track of the resource usage. The network bandwidth from the source to every other node is also tracked.

A good monitoring solution should be able to support data collection at a good frequency and do this for a long period of time without human intervention [17]. Also, the monitoring framework should not perturb the behaviour of the system being monitored [18]. The monitoring should system run as a background task without enhanced priority, ensuring that it does not interfere with the foreground tasks.

Figure 4.1 shows the overall design of our monitoring framework. Our monitoring system has five components:

1. *Container Monitor* – to log the resource usage (CPU and memory) of podman containers running on a node.

2. *Host Monitor* – to track the runtime statistics (CPU and memory availability) of the node on which it is running.

3. *Host Liveness* – to identify if any of the nodes have gone offline.

4. *Network Bandwidth Monitor* – to measure the bandwidth available between the source node of the container and other nodes.

5. *Data Collection Agent* – to collect the logs generated at other nodes in the network.

In this chapter, we describe each of these components in more detail.

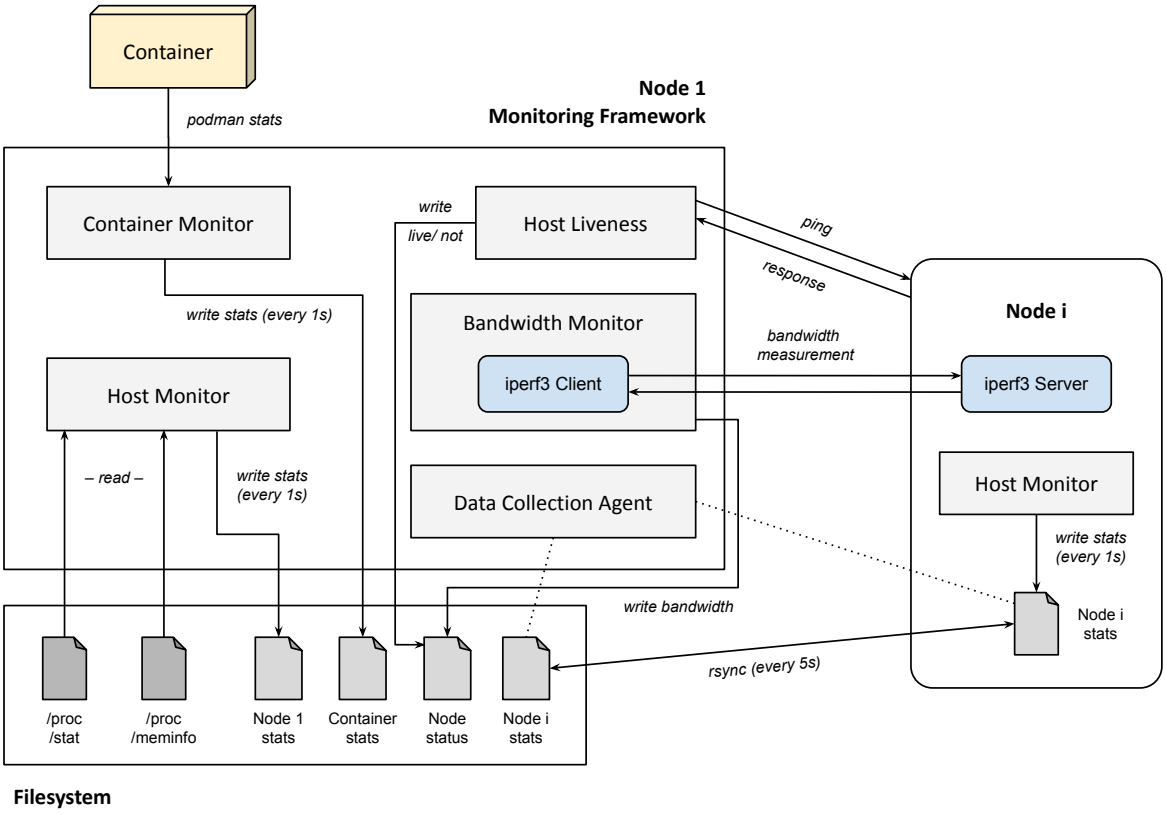

Figure 4.1: Overall design of the monitoring framework

4.1 CONTAINER MONITOR

The container monitoring agent collects and records runtime statistics for podman containers running on the host. The metrics collected by this module are the percentage CPU usage (i.e. percentage of the host's CPU consumed by the container) and the memory usage. These metrics are obtained once every second by performing a call to the podman `stats` API [19] that prints out detailed information about several runtime metrics of all running containers. We extract the relevant information for the target container from this output.

In order to handle multiple containers, we initiate parallel instances of the call. At any point in time, we have only one thread for each podman container running on that node. For every new container, information starts getting recorded automatically as soon as the container is launched. The thread continues to record the information till the container has exited or is removed.

The metrics collected are written to files, one for each container. The CPU and memory usages are reported with a timestamp indicating when the data was collected. This allows the prediction framework to train a model of the variation of the resource usage with time.

4.2 HOST MONITOR

The purpose of this module is to record the resources available on a node. Like the container monitor, this module tracks only the CPU and the memory. The host monitor retrieves the relevant runtime statistics by accessing specific control files present in the root file system of the node.

The `/proc/stat` control file records the number of CPU cycles spent by the processor doing different kinds of work - user mode, kernel mode, idle time, I/O wait times, interrupt handling times, etc. The total cycles consumed is computed as the sum total of the cycles spent for each kind of work. The values recorded in the file are the cycles spent since the last boot. So, the cycles spent doing a particular kind of work in 1 second can be estimated by differencing the values collected in 2 successive iterations.

The idle CPU on the host at any time t is calculated as follows:

$$(\%\, CPU\ idle)_t = \frac{(cycles_{idle})_t - (cycle_{idle})_{t-1}}{(cycles_{total})_t - (cycles_{total})_{t-1}} \times 100 \tag{4.1}$$

where $(cycles_{idles})_i$ and $(cycles_{total})_i$ are the idle cycles and total number of cycles consumed till time instant i. The CPU availability can then be computed as,

$$(\%\, CPU\ available)_t = 100 - (\%\, CPU\ idle)_t \tag{4.2}$$

Similarly, the memory information is available in the `/proc/meminfo` control file. Memory available for use by new processes can be read from the `MemAvailable` field in this file.

The host monitor collects and records the CPU and memory statistics once each second. These values are recorded with timestamps for use by the analysis engine.

4.3 HOST LIVENESS

The liveness module provides a mechanism for monitoring the other nodes' health and reporting them to the analysis engine for use in determining potential destinations for migration.

It checks if the other nodes in the network are online or offline. In order to do this, the source node pings other nodes, once per second. One packet of size 64 bytes (56 bytes of data + 8 bytes ICMP header) is sent to each node during the process. If the other node does not respond to the ping within a time frame of 0.1 s, it is assumed to be offline. The status of each node is written to a file, for later use.

4.4 NETWORK BANDWIDTH MONITOR

The network bandwidth plays a crucial part in determining the speed at which the migration happens. So, it must be considered while making migration decisions. The monitoring framework computes and records the bandwidth available between the source and every other node in the network.

The bandwidth available between two nodes is tracked as shown in Algorithm 4.1. The `iperf3` tool [20] is used to estimate the bandwidth. Each node in the network runs a server that listens for connections from clients on port 12345. The source node creates a new `iperf3` client object and tries to establish a connection to the `iperf3` server running at the target node. Once a connection is set up, test data is sent to the server (to measure the upload speed of the client). The result of this test is available in json format and has information about the number of bits sent out every second. This is used to estimate the bandwidth between the source and other nodes as:

$$Estimated\ bandwidth = \frac{bits\ per\ second\ (sent)}{8 \cdot 1024 \cdot 1024}\ MB/s \tag{4.3}$$

Algorithm 4.1: Bandwidth Monitor

Input: File with IP addresses of nodes
hosts ← IP addresses of online hosts
for *host* ∈ *hosts* **do**
    client ← iperf3 client
    set up connection to server at *host* : 12345
    result ← *client.run*()
    extract bits per second from *result* json
    bandwidth ← $\frac{bits\ per\ second}{8 \cdot 1024 \cdot 1024}$ MB/s
    write *bandwidth* to file
end

4.5 DATA COLLECTION AGENT

The host monitor running on every node computes the CPU and memory availability and writes these values to a file. These data points have to be shared with the source node so that it can make an informed choice about the migration targets. This is done by the data collection agent.

This module is run at the source node, once every 5 seconds. Since the host monitor records 1 reading every second, we have 5 new resource availability samples generated at each node in the network. Sending the full statistics file in every cycle is a wastage of network bandwidth and is not a good option. Instead, the source node maintains an extended history of the resource availability for each node in the network. We send only the 5 most recent values to the source node in each update cycle. This is accomplished using the `rsync` utility [21] that allows us to sync local and remote copies of files efficiently by transferring only the differences between them.

Conclusion. In this chapter, we presented the design and implementation of our monitoring framework. The framework showed how we could log the resource consumption of the container and hosts in the system. We also showed how the status of the nodes in the network could be monitored. This answers the first research question raised in Chapter 1.

CHAPTER 5: PREDICTION FRAMEWORK

Forecasting of resources is important as it allows us to schedule the containers more effectively. For example, consider a scenario where there is a container $\mathcal{C}$ (with strict timing requirements) running on host $\mathcal{H}$. Assume $\mathcal{H}$ gets overloaded after some time. If we are able to predict this ahead of time, we can migrate $\mathcal{C}$ to a different node before there is a shortage of computational resources.

Current techniques for migration perform schedulability analysis using the utilisation values of the system at the instant the analysis is performed. In reality, the utilisation values might have changed by the time migration happens. For example, if the utilisation of the destination machine goes up during that time, it may not have enough resources to accommodate the incoming container. Therefore, it is better if we predict the values of utilisation after migration and check schedulability using these estimates.

The resource usage of systems can vary rapidly. As a result, it is challenging to predict the resource utilization accurately. We use two models, *ARIMA* and *least squares curve fitting* for forecasting resource availability. The ARIMA model takes into the account the extended history of resource availability. Curve fitting, on the other hand, is trained on recent samples and focuses on recent trends in the data.

5.1 ARIMA MODEL

The monitor records the timestamps along with each observation. The resource statistics file is pre-processed to translate these timestamps to the time elapsed since the first reading. The ARIMA model works only for equally-spaced data points. Even though, the statistics are collected each second, there might be minor delays while writing them to the file system, resulting in unevenly-spaced samples. In order to resolve this, we round the time elapsed to the nearest natural number. Any missing data points are estimated by interpolation, i.e. if we have a reading x_1 at time t and x_2 at time $t+2$, then the missing value at time $t+1$ is estimated as $\frac{x_1+x_2}{2}$.

5.1.1 Grid Search

We perform a grid search to find suitable values for the three hyperparameters: p, d and q. The goal of this optimization is to find parameters that result in the best performing model. First, we define a search space $P \times D \times Q$ (3-dimensional grid). There are two versions of

the grid search we use:

- ***Grid Search with Walk-Forward Validation.*** For each 3-tuple (p, d, q) in the grid, we train an ARIMA model on 80% of the samples present in the dataset. We use the remaining 20% for walk-forward validation. Root Mean Square Error (RMSE) is used to compare the models. Algorithm 5.1 shows how we do walk-forward validation. The trained ARIMA model is used to make a 1-step prediction. The residual (prediction) is computed and used to update the squared error. In each successive iteration, we move one element from the test set to the training set and re-train the ARIMA model. This continues until we exhaust the initial test set. The RMSE is computed from the accumulated squared error as:

$$RMSE = \sqrt{\frac{accumulated\ squared\ error}{0.2 \cdot N}} \tag{5.1}$$

  where N is the number of samples in the original data set.

  Once the grid search has finished running, we get the RMSE values for all models in search space. We select the model with the lowest RMS error and use it to make forecasts about the data.

Algorithm 5.1: Walk-Forward Validation for ARIMA(p,d,q)

Input: Host stats file, p, d, q
Output: RMSE
$data \leftarrow$ read input file
$N \leftarrow$ length of $data$
$test \leftarrow data[0.8N : N]$
Squared error, $SE \leftarrow 0$
for $i \leftarrow 0$ **to** $0.2N$ **do**
  Fit $ARIMA(p, d, q)$ on $data[0 : 0.8N + i]$
  $\hat{x} \leftarrow$ Predicted value at the next step
  $SE \leftarrow SE + (\hat{x} - test[i])^2$
end
$RMSE \leftarrow \sqrt{\frac{SE}{0.2N}}$
return $RMSE$

- ***Grid Search without Walk-Forward Validation.*** For each 3-tuple (p, d, q) in the grid, we train an ARIMA(p,d,q) model on the entire dataset. Akaike Information

Criterion (AIC) [22] is used as the metric to evaluate the models. AIC is a metric that estimates how well the model fits the training data. It uses a model's log-likelihood to evaluate the quality of time-series models. It is computed as:

$$AIC = -2\ln(L) + 2k \tag{5.2}$$

where L is the maxima of the likelihood of the model and k is the number of parameters in the model. A model with a lower AIC is a better estimator of the data than another one with a bigger AIC. Once the grid search has successfully completed, we select the model with the lowest AIC value.

5.1.2 Reduced Hyperparameter Search

Performing a search for 3 parameters in a reasonably sized 3-dimensional grid is time-consuming (Table 7.4). In order to do it faster, we reduced the search space by estimating the order of differencing and the lag order. Once we have values for p and d, the problem simplifies to a 1-dimensional search for q, which is faster (shown in chapter 7).

We estimate the order of differencing (d) as shown in Algorithm 5.2. d is the minimum number of times the data has to be differenced to make it stationary.

Algorithm 5.2: Finding the Degree of Differencing

Input: Time series
Output: Degree of differencing (d)
$\alpha \leftarrow 0.05$
$d \leftarrow 0$
while $True$ **do**
  $result \leftarrow$ ADF(series)
  $p\_val \leftarrow result[1]$
  if $p\_val \leq \alpha$ **then**
    break
  end
  difference the series
  $d \leftarrow d + 1$
end
return d

We use the Augmented Dickey Fuller (ADF) test [23] to test the null hypothesis that our

data has a unit root (in other words, not stationary). If the test results in a p-value statistic smaller than our significance level, we can infer with 95% confidence that the time series is stationary. If the test indicates that the series is not stationary, we difference the data and repeat the process. The algorithm returns the value of d at which the series becomes stationary for the first time.

The next step is to determine the lag order. We use the partial autocorrelation of the data to figure out the number of lag terms that are significant. This is done as shown in Algorithm 5.3. We use the 95% confidence intervals:

$$\alpha = \pm \frac{2}{\sqrt{N}} \tag{5.3}$$

where N is the number of samples in the time series [24]. The time series is differenced d times (computed earlier) to make it nearly stationary. The partial autocorrelation of the differenced series is then computed. We consider only lags whose autocorrelation value is beyond the significance level for estimating p.

Algorithm 5.3: Finding the Lag Order

Input: Differenced time series
Output: Lag order (p)
$\alpha \leftarrow \frac{2}{\sqrt{len(series)}}$
$pac \leftarrow PACF(series)$
$p \leftarrow 0$
for $i \leftarrow 1$ **to** $len(pac)$ **do**
  if $\mid pac[i] \mid > \alpha$ **then**
    $p \leftarrow p + 1$
  else
    break
  end
end
return p

Once we have estimated the values of p and d, we use them to perform a reduced grid search. We do a 1-dimensional search for the value of q. For each q considered, we train an $ARIMA(p, d, q)$ model. Akaike Information Criterion (AIC) is used to compare the quality of the models obtained. We select the model that yields a fit with the lowest AIC.

5.2 CURVE FITTING

The first step is to identify the degrees of two polynomials that best describe the CPU and memory availability of the node. This is done as described in Algorithm 5.4. Only curves of degrees 1 to 10 are considered for this process. The complexity of the Gauss-Jordan elimination process increases with the size of the augmented matrix. Higher degree curves lead to larger augmented matrices that take more time to solve.

Algorithm 5.4: Determining the Degree of the Best Polynomials

Input: Host stats file

Output: Degrees of the best polynomials for CPU and memory stats

$data \leftarrow$ last 20 lines of the file

$train \leftarrow data[0:18]$

$test \leftarrow data[18:20]$

for $k \leftarrow 1$ **to** 10 **do**

  Fit 2 degree k polynomials on CPU and memory data in $train$

  Predict 19^{th} and 20^{th} values

  Compute RMS errors over 19^{th} and 20^{th} values (equation 5.4)

end

$k_{cpu} \leftarrow$ degree of CPU curve with least RMS error

$k_{mem} \leftarrow$ degree of memory curve with least RMS error

return k_{cpu} and k_{mem}

20 most recent data samples (pre-processed) are read from the file. We use 90% of these samples for fitting the curve. The remaining 10% (2 samples) are used for testing. We use a $90-10$ split, because the curve fitting model is used to account for recent trends in the resource availability. By including 90% of these samples in the training set, we make sure the model fits the recent data points reasonably well.

For each degree k, the coefficients of the best fit polynomial are obtained using the approach described in chapter 2. This polynomial is then used to predict the next 2 values - $\hat{x_{19}}$ and $\hat{x_{20}}$. Root Mean Square (RMS) error is used to compare the accuracy of the models obtained. We select the polynomial with the lowest root mean square error and use its degree to estimate the future data points. If x_{19} and x_{20} (in $test$) are their actual values, the RMS error for this degree k polynomial is computed as:

$$RMSE = \sqrt{\frac{(x_{19} - \hat{x_{19}})^2 + (x_{20} - \hat{x_{20}})^2}{2}} \tag{5.4}$$

This is done for both CPU and memory values. The degrees output by Algorithm 5.4 (k_{cpu} and k_{mem}) are then used to fit two polynomials on the 20 most recent CPU and memory data values. Let

$$f_1(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + ... + a_{k_{cpu}} t^{k_{cpu}} \tag{5.5}$$

be the best curve for the CPU availability, and let

$$f_2(t) = b_0 + b_1 t + b_2 t^2 + b_3 t^3 + ... + b_{k_{mem}} t^{k_{mem}} \tag{5.6}$$

be the corresponding curve for the memory data. The CPU and memory availability at some time τ are estimated as $f_1(\tau)$ and $f_2(\tau)$ respectively.

Conclusion. In this chapter, we presented two approaches used for fitting the data collected by the monitoring framework. The models obtained after learning can be used to predict the state of the system. This answers the second research question (Chapter 1) and validates the part of the hypothesis related to forecasting the resource availability of the nodes in the network.

CHAPTER 6: ANALYSIS ENGINE

The analysis engine uses the resource statistics collected by the monitoring framework to make all migration-related decisions. It carries out three main functions:

1. maintain a ranked set of potential destination nodes that have enough resources to host the container.
2. determine if and when there is a need to migrate the container
3. when there is a need for migration, ensure that the chosen nodes still have enough resources for the container to meet its computational and timing requirements.

In this chapter, we describe why each of these functions is important and how the analysis engine uses the information to accomplish these tasks.

6.1 RANKING POTENTIAL DESTINATIONS

Before selecting a node as a potential destination for migration, we need to make sure that it has enough resources to meet the resource demands of the container. We use the notion of eligibility sets [25] to refer to the subset of nodes that satisfy the above condition. Algorithm 6.1 shows how we create the eligibility set.

The data collection agent collects the resource availability information from the other nodes in the network. We use the data collected by the agent to train ARIMA models for predicting the future availability on these nodes. Let $\hat{x}$ and $\hat{y}$ be the estimated CPU and memory availability of a node. The resource availability might change rapidly over time. We consider the standard deviation over the recent samples to reduce the impact of recent fluctuations on the estimated availability [25]. If σ_1 and σ_2 are the standard deviations of the CPU and memory availability (computed over the last 10 readings), the node is considered an eligible target for migration if:

$$(\hat{x} - \sigma_1 > r_{cpu}) \wedge (\hat{y} - \sigma_2 > r_{mem}) \tag{6.1}$$

Here r_{cpu} and r_{mem} are the maximum CPU and memory needs of the container on that node.

Once we know the set of nodes that have enough resources to run the container, we create a ranked list of targets. We use the bandwidth information collected by the network bandwidth monitor to rank the nodes in the eligibility set. The nodes are ranked in the descending order of the bandwidth between the source (running the container) and that

Algorithm 6.1: Finding Eligible Migration Targets

Input: Node stats file
Output: List of eligible destinations
$hosts \leftarrow$ IP addresses of online hosts
$E \leftarrow \phi$
for $host \in hosts$ **do**
  $data \leftarrow$ read $host$'s resource statistics file
  train an ARIMA model on $data$ (reduced hyperparameter search)
  $\hat{x}, \hat{y} \leftarrow$ predicted CPU and memory availability
  $\sigma_1, \sigma_2 \leftarrow$ standard deviation over last 10 CPU and memory values
  if $(\hat{x} - \sigma_1 > r_{cpu}) \wedge (\hat{y} - \sigma_2 > r_{mem})$ **then**
    $E \leftarrow E \cup \{host\}$
  end
end
return E

node. Nodes with higher network bandwidth are preferred to minimize the time taken to transfer the dumped memory images to that node.

The ranked list of nodes obtained is written to a file for use by the migration executor. This component is run once every 10 seconds to produce a new list of eligible targets. The file is then updated to reflect the new eligibility information.

6.2 TRIGGER FOR MIGRATION

If we want to keep a container running smoothly, we have to initiate migration before we reach a scenario where the current node can no longer sustain its resource needs. This can be achieved by repeatedly forecasting the resource availability on the source node and checking if it falls below the requirement.

To do this, we run the curve fitting algorithm on the 20 most recent resource statistics recorded by the host monitor running at the source node. The algorithm is run on the last 20 samples to account for recent trends in the resource availability. Once the degrees of the best fitting polynomials (for CPU and memory) are obtained, we use them to predict the resource availability 2 seconds into the future. This is a safe margin that allows us enough time to migrate and restore the container at the destination. Like the other nodes, we also use the standard deviation over 10 most recent values to mitigate the effects of fluctuations

on our predictions.

If $\hat{x}$ and $\hat{y}$ are the CPU and memory estimates; σ_1 and σ_2 are the standard deviations of the CPU and memory availability; and r_{cpu} and r_{mem} are the container's resource needs, we trigger the migration if:

$$(\hat{x} - \sigma_1 \leq r_{cpu}) \vee (\hat{y} - \sigma_2 \leq r_{mem}) \tag{6.2}$$

This process is repeated once every second on 20 recent data samples recorded by the host monitor. This includes the most recent sample recorded by the host monitor running on that node. If the aforementioned condition fails at any point, it indicates that the current host node would not be able to meet the application's resource requirements in the future, in which case we initiate migration.

6.3 VALIDATING THE ELIGIBILITY SET

The eligibility sets are created by running the ARIMA model once every 10 seconds. The trigger for migration might arrive anytime before the next invocation of the ARIMA model. The data collection agent collects the runtime statistics from the other nodes, once every 5 seconds. So, depending on when the trigger arrives, we may or may not have received a new batch of resource availability information. We consider 2 cases:

- ***Case 1:*** *A new batch of data has been collected from the other nodes*

  The status of the nodes might have changed since we created the eligibility set. As a result, the ranked lists of targets might be outdated. For each node, the curve fitting algorithm is run on the 20 most recent data samples. We use the faster curve fitting approach (instead of the ARIMA model) because this validation step is in the critical path of the migration. The degrees of the best fit curves for CPU and memory are computed and used to forecast the resource availability. We use these predictions and the new standard deviations to recompute the eligibility set (as described in section 6.1). The nodes are ranked again to reflect their latest status.

- ***Case 2:*** *No new data samples have been collected*

  In this case, the nodes in the eligibility set are still assumed to have resources to run the container. However, the status of the network might have changed. For nodes in the eligibility set, we check if they are online by analyzing the output of the liveness monitor. Nodes that have gone offline are removed from the eligibility set. The online nodes are re-ranked based on the latest bandwidth information collected by the network bandwidth monitor.

The newly created ranked list is then used by the migration executor. It selects a subset of these eligible nodes as potential destinations for migration. The application is checkpointed at the source node and the dumped memory pages are copied to each of the selected nodes. This is done to ensure that the migration is fault-tolerant. Even if any of the potential targets crash or a part of the network fails, migration proceeds to completion elsewhere. The migration executor is notified when any migration attempt succeeds. The pending attempts are killed to free the system resources.

Conclusion. In this chapter, we described how the analysis engine uses the forecasts for selecting potential targets and triggering the migration. This validates the part of the hypothesis related to the use of forecasts from the predication framework. Section 6.1 of this chapter answers the third research question raised in Chapter 1.

CHAPTER 7: EVALUATION

7.1 SYSTEM SPECIFICATIONS

To evaluate the different components of our system, we conducted experiments using the nodes listed in Table 7.1. The system consists of 5 nodes with different CPU and memory capacities. The nodes are Ubuntu 20.04 virtual machines running on two servers.

Table 7.1: System Specifications

Host	**CPU(s)**	**Memory**	**Operating System**	**Processor Type**
H1	8	48 GB	Ubuntu 20.04	Intel Xeon E3-1270 v6 at 3.80 GHz
H2	4	12 GB	Ubuntu 20.04	Intel Xeon E3-1270 v6 at 3.80 GHz
H3	4	16 GB	Ubuntu 20.04	Intel Xeon E3-1220 v5 at 3.00 GHz
H4	4	6 GB	Ubuntu 20.04	Intel Xeon E3-1270 v6 at 3.80 GHz
H5	2	4 GB	Ubuntu 20.04	Intel Xeon E3-1220 v5 at 3.00 GHz

We used the `progrium/stress` image [26] to launch a podman container on node H1 (i.e., H1 is the source node). The container has 4 CPU stressors and 2 memory workers running inside it.

7.2 CONTAINER RESOURCE NEEDS

A container might need different amounts of resources on different nodes. The number of CPU cycles consumed might vary depending on underlying architecture. For example, if there are more cache misses on a machine, the cycles needed to finish execution might be more than on another machine with fewer cache misses.

We assume that the maximum resource usage of the container on each node is known ahead of time. We have a closed system where no new nodes enter the system. For each node in our system, we launched an instance of the container and used the container monitor to collect 15 minute traces of the resource usage.

The observed mean, standard deviation (SD) and maximum value of the CPU and memory usages are listed in Table 7.2. The analysis engine uses the maximum usage values listed here while making migration decisions.

Table 7.2: CPU and Memory Usage of the Container on Each Node

Host	**CPU Usage (%)**			**Memory Usage (MB)**		
	Mean	**SD**	**Max**	**Mean**	**SD**	**Max**
H1	5.499	2.59	9.98	133.477	52.215	256.8
H2	4.92	2.351	9.92	130.212	54.145	258.4
H3	5.457	2.602	9.98	130.125	54.199	257.7
H4	5.49	2.609	10	130.525	54.029	255.8
H5	5.526	2.603	10	127.086	56.223	258.1

7.3 LOAD CONFIGURATIONS

The nodes in the system may have a variety of different tasks running on them, consuming different amounts of hardware resources. In order to simulate some workloads, we used the `stress-ng` workload generator [27]. `stress-ng` has configurable parameters that allows us to impose different CPU and memory stress on the nodes. We used 10 different load configurations for evaluation. The purpose of these configurations is to obtain resource statistics with different trends, which in turn allows us to evaluate if our prediction framework can make good forecasts for the resource availability. The configurations are listed in Table 7.3. In the first configuration, we have 4 instances of the CPU stressors spinning on `sqrt()` and 2 memory workers writing 256 MBs each. In configuration 2, we have an additional load on the system towards the end of the observation period. In configurations 7 and 8, stress is only applied at certain times. For the last 2 configurations, we used the `stress-ng` matrix stressors that perform several matrix operations on large floating point arrays.

These loads were run on node H2. For each load configuration, we ran the host monitor and collected the resource availability statistics. The data collection agent on node H1 obtains the data recorded by the host monitor running on H2. The raw data is processed to obtain samples with equally-spaced time gaps.

7.4 ARIMA MODEL

7.4.1 Comparing the Model Selection Methods

To evaluate the performance of the ARIMA model, we ran it on 10 minute traces obtained from node H2. We tried three different methods for model selection to compare their accuracy and run times.

Table 7.3: Load Configurations

	stress-ng Load			
	CPU Workers	**Memory Workers**	**MB Written per Memory Worker**	**Configuration**
1	4	2	256	default
2	4	2	256	additional load for the last 4s
3	2	4	256	default
4	3	6	256	default
5	6	3	256	default
6	4	8	512	default
7	4	2	256	alternating load/ no load (1min period)
8	2	2	256	alternating load/ no load (random times)
9	1 instance of matrix stressor			
10	2 instances of matrix stressor			

- ***Grid Search with Walk-Forward Validation.*** We did a $5 \times 3 \times 4$ grid search to find a good set of parameters for the ARIMA model. For each 3-tuple (p, d, q), the ARIMA model is trained on 80% of the samples in the data file. The remaining 20% are used for walk-forward validation. The model is iteratively refined by adding a new element to the training set in every cycle as shown in Algorithm 5.1. We find the model with the lowest value of the RMS error.

- ***Grid Search without Walk-Forward Validation.*** The model is trained on the entire dataset. Akaike Information Criterion (AIC) is used as the metric to decide which model suits the data best. We perform a $5 \times 3 \times 4$ grid search and identify the model with the lowest AIC.

- ***Reduced Hyperparameter Search.*** We follow the approach discussed in the chapter 5. We first determine the order of differencing using the ADF test in Algorithm 5.2. The lag order p is determined from the partial autocorrelation (PAC) of the data as described in Algorithm 5.3. Figure 7.1 is the PAC plot showing the partial autocorrelation values for the first 20 lags of the CPU availability data from node H2 (under load configuration 1). The X-axis is the lag and the Y-axis is the PAC value. The significance level for this dataset is computed using Equation 5.3. The PAC of lag 0 is

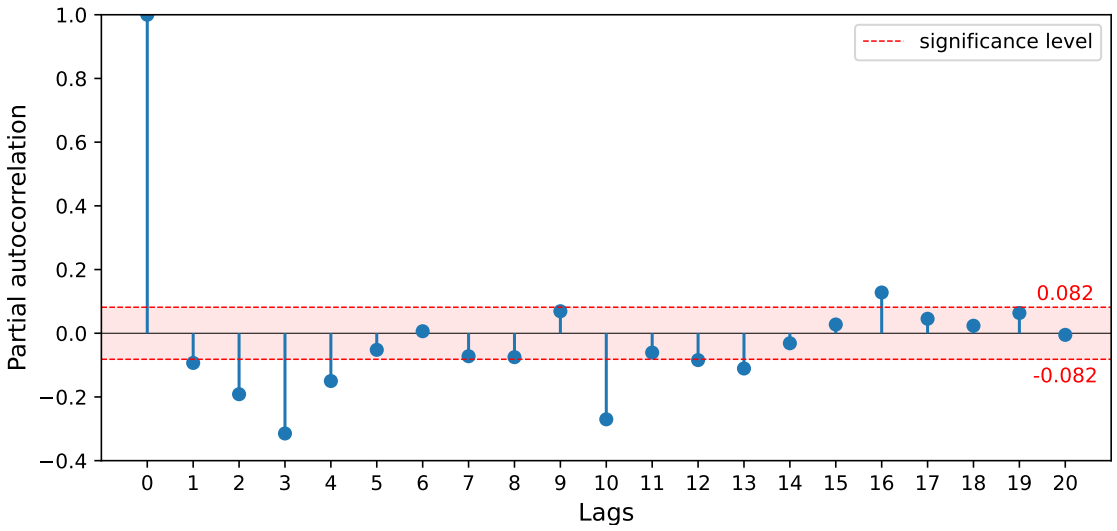

Figure 7.1: Partial autocorrelation plot for the CPU available on node H2 (under load 1)

Table 7.4: Training the ARIMA Model on CPU Data from Node H2

Method	**Search Space**			**Best Params**	**Time (s)**
	p	**d**	**q**		
Walk-Forward Validation, Minimizing RMSE	0,1,2, 3,4	0,1,2	0,1,2, 3	p: 0, d: 1, q: 3	1178.169
No Walk-Forward Validation, Minimizing AIC	0,1,2, 3,4	0,1,2	0,1,2, 3	p: 3, d: 1, q: 3	11.819
Reduced Search (for q), Minimizing AIC	-	-	0,1,2, 3,4	p: 4, d: 1, q: 3	1.772

always 1. The next 4 lags are below the lower significance level −0.082 (shown in red). Lag 5, however, is not significant. So, the lag order for this time series is taken to be 4. For q, we do a reduced 1-dimensional grid search. The models obtained during this process are ranked based on their AIC value. Lower the value of the AIC, the better the model.

The results obtained on training the ARIMA using these 3 methods are summarized in Table 7.4. The time recorded in the table is the total time taken for finding the best parameters. It is observed that performing a grid search with walk-forward validation is computationally expensive and takes about 20 minutes for a dataset containing 600 data points. Without walk-forward validation, the optimal model can be selected about 100 times faster. The model selection is even quicker when we do a reduced grid search - it took

Table 7.5: RMSE and AIC for the Models Selected by the 3 Methods

Model	**RMSE**	**AIC**
ARIMA(0,1,3)	10.105	3686.441
ARIMA(3,1,3)	10.521	3667.859
ARIMA(4,1,3)	10.76	3669.609

only 1.772 seconds to find the best model for the CPU data.

In order to compare the accuracy of the models selected by these 3 methods, we computed their root mean square error and AIC. The experiment is performed using the CPU data from node H2, under load 1. For finding the RMSE, the models are trained on 80% of the data. The remaining 20% is used in the computation of the RMSE. The results are shown in Table 7.5. It is observed that there is no large disparity between the accuracies of the 3 models. The first approach yields the model with the lowest RMSE. Among these 3 models, the one chosen by the reduced grid search had the highest RMSE.

The performance of the model selected by the reduced grid search is shown in figure 7.2. The X-axis shows the data samples and the Y-axis is the CPU availability. The model is trained on 80% of the samples and tested on the remaining 20%. The blue, orange and green lines represent the training data, test data and predictions respectively. It is observed that the predictions follow the test data closely. However, the CPU data fluctuates a lot resulting in a worse RMSE. 3-dimensional grid search results in a model with a lower root mean square error, but the algorithm takes significantly more time to complete. In other words, we tradeoff a small amount of accuracy for a much better run time.

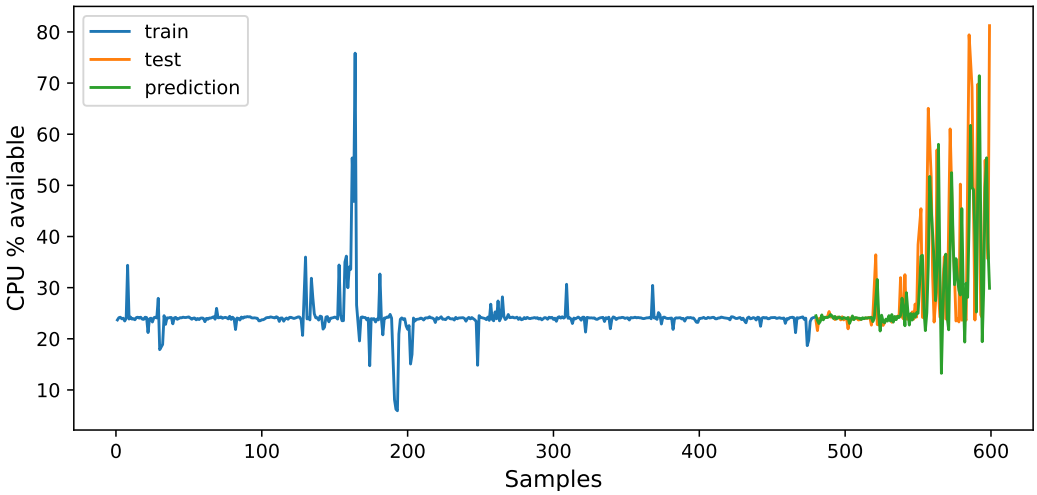

Figure 7.2: Plot showing the performance of the ARIMA(4,1,3) model (load 1)

7.4.2 Estimating the Total Time Taken for CPU and Memory Data

In order to estimate the average time taken for predicting both CPU and memory availability of H2, we performed a reduced grid search on 10 traces collected from H2. For each trace, we ran the experiment 20 times to mitigate the impact of fluctuations in run times due to external factors. The results of this experiment are shown in Table 7.6. It is observed that reduced grid search for CPU and memory data takes about 2.713 seconds (average over 10 traces).

Table 7.6: Results Obtained on Running the Reduced Grid Search on 10 Minute Traces

Load	**CPU Available**		**Memory Available**		**Avg. Time (s)**
	Model	**AIC**	**Model**	**AIC**	
1	(4,1,3)	3669.609	(2,0,1)	13410.081	2.499
2	(6,1,1)	2292.084	(4,1,0)	14060.276	4.977
3	(1,0,4)	1789.687	(3,0,0)	13354.52	1.724
4	(1,0,0)	-103.119	(3,0,4)	12843.014	1.824
5	(3,0,0)	-553.191	(2,0,4)	13001.752	3.069
6	(1,0,0)	-844.225	(4,1,0)	13356.485	2.911
7	(2,0,2)	4154.31	(1,0,3)	14309.965	2.17
8	(2,0,3)	3848.253	(2,0,4)	14523.773	3.926
9	(3,0,0)	2930.596	(1,0,0)	13293.29	1.883
10	(1,0,0)	2465.667	(1,0,1)	12636.381	2.15
					***Mean*: 2.713**

7.5 CURVE FITTING

The curve fitting algorithm was run on the last 20 samples from 10 minute traces generated at H2. Table 7.7 provides a summary of the results obtained in the process. In addition to the Root Mean Square Error (RMSE), the table lists the Root Mean Square Percentage Error (RMSPE) for the memory data. Let $\hat{x_{19}}$ and $\hat{x_{20}}$ be the predictions. The RMSPE is computed over the last 2 values (say, x_{19} and x_{20}) as:

$$RMSPE = \sqrt{\frac{(\frac{x_{19}-\hat{x_{19}}}{x_{19}})^2 + (\frac{x_{20}-\hat{x_{20}}}{x_{20}})^2}{2}} \tag{7.1}$$

For each trace, we ran the experiment 20 times and computed the average time over

Table 7.7: Results Obtained on Fitting 2 Curves on 20 Recent CPU and Memory Samples

Load	**CPU Available (%)**		**Memory Available (KB)**			**Avg. Time (ns)**
	Deg.	**RMSE**	**Deg.**	**RMSE**	**RMSPE**	
1	1	22.512	7	816.774	0.000019	298300
2	1	2.926	9	30916.63	0.000714	304167
3	2	0.051	1	364.43	0.000008	282833
4	1	0	1	112.616	0.000003	333058
5	1	0	2	746.913	0.000017	283806
6	2	0	3	69.253	0.000002	279172
7	2	13.52	2	35706.169	0.000831	328186
8	1	0.056	2	260.489	0.000006	294470
9	1	0.266	1	2780.728	0.000064	287055
10	9	2.915	1	33328.497	0.000772	296233
						***Mean*: 298728**

these 20 runs. For configurations 4, 5 and 6, the node was overloaded and 0% of CPU was available for use by new processes. The curve fitting algorithm predicted this trend accurately, resulting in no error.

It is observed that fitting 2 curves for CPU and memory data takes about 298728 nanoseconds (average over 10 traces). This is significantly faster than the ARIMA model.

7.6 RANKING POTENTIAL DESTINATIONS

We set up four different scenarios to check if the node ranking algorithm works as intended. For all scenarios, data collected over 10 minutes was used for training the ARIMA models. A 1-step prediction (1 second ahead) is made and used along with the standard deviation to determine the eligibility set. Nodes in the eligibility set are ranked on the basis of the bandwidth available. `stress-ng` [27] has been used to generate different loads on the hosts.

The four scenarios are:

- ***Scenario 1.*** H2, H3, H4 and H5 are online and have enough resources to host the container. It is observed that all 4 nodes are part of the eligibility set. Also, the nodes are seen to be ranked in the descending order of their available bandwidth, with H3 being the best target.

- ***Scenario 2.*** Scenario 2 has been set up similar (loads) to Scenario 1, except for

that fact that one of the nodes is offline and does not communicate with the source node. H2, H4 and H5 are online and have enough resources for container. H3 is offline and so, the source node H1 does not have any information about its CPU or memory availability. We observe that H3 is not part of the eligibility set. The other nodes are ranked as expected.

- ***Scenario 3.*** All 4 nodes are online. However, node H4 is overloaded and does not have enough CPU available to host the migrating container. So, it is not part of the eligibility set for migration. The other 3 nodes have enough resources to run the container and so, are part of the eligibility set. H2, H3 and H5 are ranked according to their bandwidths.

- ***Scenario 4.*** H2 is offline. The other 3 are online. However, H5 is fully overloaded and has no spare CPU available. It is observed that only 2 nodes - H3 and H4 are part of the eligibility set and they are ranked accordingly.

Table 7.8 presents the results of this experiment. The resource availability values listed in the table are the estimates (obtained by running the ARIMA model) after 1 second. The table also contains the standard deviations computed over the last 20 samples. The bandwidth was the value (in MB/s) recorded by the network bandwidth monitor when the experiment was run. Rank 1 corresponds to the most preferred target node, while rank 4 is the least preferred one.

For each scenario, the node selection and ranking algorithm work as expected. Nodes that are offline or overloaded are not part of the eligibility set. It is also observed that the nodes in the eligibility set are ranked according to the descending order of the bandwidth from H1 to that node. Nodes with higher network bandwidth are given a higher priority to reduce the time taken to transfer the dumped memory pages.

7.7 VALIDATING THE ELIGIBILITY SET

Node H2 was stressed with the load configuration 2. The data collection agent on node H1 collects the availability information from H2 once every 5 seconds. The ARIMA model is run on 592 samples (after interpolation) collected from H2. A 1-step prediction using the best ARIMA model suggests that the container can be run safely on H2. Over the course of the next 5 seconds, H2 is stressed with an additional load, decreasing the CPU availability. If the trigger for migration arrives after these samples are collected at H1, the curve fitting algorithm would be run on 20 recent samples.

Table 7.8: Eligibility Sets and Ranking Produced by the Analysis Engine

	Host	CPU (%)		Memory (GB)		Bandwidth	Eligible?	Rank
		Avail.	SD	Avail.	SD			
1	H2	73.554	0.316	10.274	0.001	70.722	✓	2
	H3	32.678	3.615	14.119	0.0004	71.883	✓	1
	H4	23.665	0.13	4.182	0.0004	69.584	✓	3
	H5	49.602	0.557	2.2	5.09e-6	69.169	✓	4
2	H2	73.722	0.331	10.266	0.0001	63.898	✓	3
	H3	-	-	-	-	offline	✗	-
	H4	23.821	0.259	4.169	0.0001	73.404	✓	2
	H5	49.04	0.271	2.136	0.0001	73.784	✓	1
3	H2	23.907	0.631	10.25	0.0001	71.963	✓	2
	H3	73.686	0.278	6.706	0.092	71.181	✓	3
	H4	0.127	0.451	4.155	0.0003	71.302	✗	-
	H5	49.078	0.246	2.185	4.83e-6	74.178	✓	1
4	H2	-	-	-	-	offline	✗	-
	H3	73.977	0.309	13.854	0.012	75.691	✓	1
	H4	48.27	0.113	3.656	0.0006	74.631	✓	2
	H5	0	0	2.646	0.01	74.412	✗	-

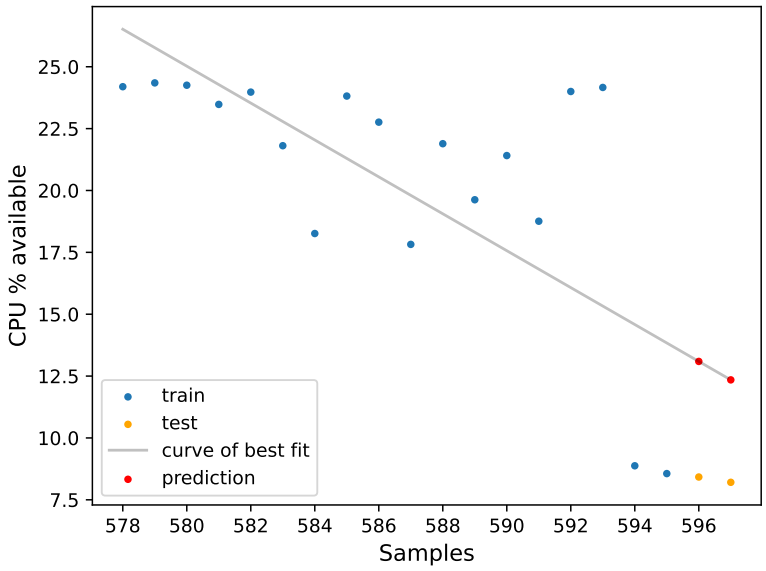

Figure 7.3: Plot showing the best fit curve obtained from curve fitting

Figure 7.3 shows the recent CPU usage values and the best fit curve. The CPU usage is predicted using this best fit curve and used to check if H2 is capable of hosting the container. The validation module figures out that H2 does not have enough CPU available for the container and so, removes it from the eligibility set. The results of this experiment are summarized in Table 7.9.

Table 7.9: Eligibility of Node H2 per the ARIMA and Curve Fitting Models

Model	**Samples**	**CPU (%)**		**Memory (GB)**		**Bandwidth**	**Eligible?**
		Avail.	**SD**	**Avail.**	**SD**		
ARIMA	1-592	23.054	2.265	10.002	0.0003	68.421	✓
Curve Fitting	578-597	11.601	5.821	10.343	0.154	71.421	×

CHAPTER 8: RELATED WORK

There have been a wide range of efforts aimed at supporting live migration of virtual machines in large cloud data centers. The paper by Choudhary et al. [28] provides a comprehensive review and comparison of several live VM migration techniques using performance metrics like migration time, application downtime and the amount of data transferred as a result of the migration. Container migration efforts are far fewer in number.

Govindaraj et al. [29] proposed a live migration scheme called redundancy migration and demonstrated it using linux containers (LXC, LXD) for factory automation applications. The redundancy migration scheme proposed in the paper includes a replay phase where the destination node replays the packets after the checkpoint, to catch up to the state of the application at the source. This approach has a lower downtime than stock LXD migration.

Sinha et al. [30] proposed a new live migration approach for an intra-host scenario where the source and destination nodes are virtual machines residing on the same physical device. This paper exploits the fact that the memory pages for the two containers co-reside on the same host and relocates the ownership of these pages, instead of actually copying them over the network. They show that this approach reduces the overall service downtime for the application.

Souza Junior et al. [31] proposed a migration scheme that snapshots the memory and exploits the layered nature of the filesystem to transfer them over to the destination. They identified parts of the container layer that are not currently being modified (at the source) and transferred them ahead of time. They evaluated their approach on a fog computing testbed for geo-distributed applications and showed a reduction in downtime over migration without a layered approach.

Our approach differs in a few ways. First, in order to make the migration more resilient to network-related failures or system crashes (at the destination), we migrate to multiple destinations simultaneously. This is better suited for real-time applications that have strict response time requirements as there is no time lost retrying a migration. Second, we use the resource availability forecasts to select potential targets for migration. This allows the framework to account for trends in resource availability better.

CHAPTER 9: CONCLUSION

Real-time containers have strict response time requirements. When the source node of a container cannot sustain its computational needs, we can migrate the container to another node to ensure that it continues to meet the timing requirements. In this work, we presented the design and implementation of a framework for supporting reliable container migration. We have shown how it is possible to plan the migration by monitoring, forecasting and analyzing the resource availability trends of nodes in the system. There are three main components developed as part of this work: (1) a monitor to track the resources and network status, (2) a prediction framework to forecast the resources available and (3) an analysis engine to make migration-related decisions. These are important components of a real-time container migration framework. We have shown that the prediction framework can make good forecasts of the resource availability. The least squares curve fitting algorithm (on the critical path of migration) is able to process the data and predict future trends in a timely manner. Our results indicate that the analysis engine works as intended and is able to select the correct subset of nodes as destinations for migration.

Limitations and Future Work.

- In the current state, the prediction models do not perform well when the CPU and memory data fluctuate rapidly. We plan to explore other prediction strategies that can handle fluctuations better.
- The current framework assumes that the system is closed and all hosts are known in advance. We would like to expand this framework to support non-closed systems where new nodes may enter the system.
- The container monitor used in our framework supports only podman containers. In a future work, we would like to expand this to handle other container engines and evaluate our framework on a network with heterogeneous nodes.

REFERENCES

[1] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos, "Challenges and opportunities in edge computing," in *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, 2016, pp. 20–26.

[2] N. Wang, B. Varghese, M. Matthaiou, and D. S. Nikolopoulos, "Enorm: A framework for edge node resource management," *IEEE Transactions on Services Computing*, vol. 13, no. 6, pp. 1086–1099, 2020.

[3] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015, pp. 171–172.

[4] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, "Containers and virtual machines at scale: A comparative study," in *Proceedings of the 17th International Middleware Conference*, ser. Middleware '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2988336.2988337

[5] "Docker." [Online]. Available: https://www.docker.com/

[6] "Podman." [Online]. Available: https://podman.io/

[7] J. D. Hamilton, *Time Series Analysis*. Princeton University Press, 1994.

[8] P. A. Moran and P. Whittle, "Hypothesis testing in time series analysis." 1951.

[9] H. S. Uhler, "Method of least squares and curve fitting," *J. Opt. Soc. Am.*, vol. 7, no. 11, pp. 1043–1066, Nov 1923.

[10] "Curve Fitting – Method of Least Squares." [Online]. Available: http://www.notespoint.com/curvefitting-leastsquares/

[11] K. E. Atkinson, *An Introduction to Numerical Analysis*, 2nd ed. New York: John Wiley & Sons, 1989.

[12] G. W. Stewart, "Gauss, statistics, and gaussian elimination," *Journal of Computational and Graphical Statistics*, vol. 4, no. 1, pp. 1–11, 1995. [Online]. Available: http://www.jstor.org/stable/1390624

[13] "CRIU." [Online]. Available: https://criu.org/Main_Page

[14] "google/cadvisor - GitHub." [Online]. Available: https://github.com/google/cadvisor

[15] L. L. Jiménez, M. G. Simón, O. Schelén, J. Kristiansson, K. Synnes, and C. Åhlund, "Coma: Resource monitoring of docker containers," in *CLOSER*, 2015.

[16] F. A. Oliveira, S. Suneja, P. N. Shripad Nadgowda, and C. Isci, "A Cloud-native Monitoring and Analytics Framework," IBM Research Division, Technical Report, December 2017.

[17] M. Jägemar, "Utilizing hardware monitoring to improve the quality of service and performance of industrial systems," Ph.D. dissertation, Mälardalen University, 2018.

[18] C. Fidge, "Fundamentals of distributed system observation," *IEEE Softw.*, vol. 13, no. 6, p. 77–83, nov 1996. [Online]. Available: https://doi.org/10.1109/52.542297

[19] "podman stats - Ubuntu manpages." [Online]. Available: https://manpages.ubuntu.com/manpages/impish/man1/podman-stats.1.html

[20] "iPerf - The ultimate speed test tool for TCP, UDP and SCTP." [Online]. Available: https://iperf.fr/

[21] "rsync - Linux man page." [Online]. Available: https://linux.die.net/man/1/rsync

[22] H. Akaike, "A new look at the statistical model identification," *IEEE Transactions on Automatic Control*, vol. 19, no. 6, pp. 716–723, 1974.

[23] D. A. Dickey and W. A. Fuller, "Distribution of the estimators for autoregressive time series with a unit root," *Journal of the American Statistical Association*, vol. 74, no. 366, pp. 427–431, 1979. [Online]. Available: http://www.jstor.org/stable/2286348

[24] J. Shekeine, L. A. Turnbull, P. Cherubini, R. de Jong, R. Baxter, D. Hansen, N. Bunbury, F. Fleischer-Dogley, and G. Schaepman-Strub, "Primary productivity and its correlation with rainfall on Aldabra Atoll," *Biogeosciences Discussions*, vol. 12, no. 2, pp. 981–1013, Jan. 2015.

[25] V. Padhye, D. Kulkarni, and A. R. Tripathi, "Node selection for placement of migratory tasks in wide-area shared computing environments," 2010.

[26] "progrium/stress - Docker Hub." [Online]. Available: https://hub.docker.com/r/progrium/stress/

[27] "stress-ng - Ubuntu manpages." [Online]. Available: https://manpages.ubuntu.com/manpages/focal/man1/stress-ng.1.html

[28] A. Choudhary, M. C. Govil, G. Singh, L. K. Awasthi, E. S. Pilli, and D. Kapil, "A critical survey of live virtual machine migration techniques," *J. Cloud Comput.*, vol. 6, no. 1, dec 2017.

[29] K. Govindaraj and A. Artemenko, "Container live migration for latency critical industrial applications on edge computing," in *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, 2018, pp. 83–90.

[30] P. K. Sinha, S. S. Doddamani, H. Lu, and K. Gopalan, "mwarp: Accelerating intra-host live container migration via memory warping," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2019, pp. 508–513.

[31] P. Souza Junior, D. Miorandi, and G. Pierre, "Stateful container migration in geo-distributed environments," 12 2020.