Distributed Resource Provisioning for Containers Using Machine Learning and Live Migration

By:

Alejandro González Araya

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in
COMPUTER ENGINEERING

UNIVERSITY OF PUERTO RICO
MAYAGÜEZ CAMPUS

2020

Approved by:

Emmanuel Arzuaga Cruz, Ph.D. President, Graduate Committee	Date
Manuel Rodríguez Martínez, Ph.D. Member, Graduate Committee	Date
Wilson Rivera Gallego, Ph.D. Member, Graduate Committee	Date
Iván J. Baigés Valentín, Ph.D. Representative, Office of Graduate Studies	Date
Dr. Irvin J. Balaguer Álvarez, Ph.D. Chair, Department of Electrical and Computer Engineering	Date

ABSTRACT

Cloud computing uses pools of virtual machines to provide shared computing resources. Provisioning and management of these resources are usually done using statistical algorithms to help decide how to better utilize available compute power. Recently, this has been performed mostly by using live migration of virtual machines. Nowadays containers provide the flexibility to handle many software environments and tasks in a lighter-weight virtualization scheme, providing a more agile alternative to virtual machines for certain applications. Application checkpointing coupled with a container manager allows the live migration of a container.

In this thesis, we use container live-migration and real-time monitoring to develop a cloud resource provisioning platform that enables an improvement in usage and execution of containers. As a result, the available resources can be tuned and distributed in a more efficient manner, providing a better use of cloud. This research explores the use of container live migration techniques to improve cloud resource provisioning. We present the design and development of our container live-migration, monitoring and provisioning toolset as well as a performance evaluation and characterization.

RESUMEN

La computación en la nube utiliza grupos de máquinas virtuales para proporcionar recursos informáticos compartidos. El aprovisionamiento y la gestión de estos recursos se realizan generalmente mediante algoritmos estadísticos para ayudar a decidir cómo utilizar los recursos de cómputo disponibles de una mejor manera. Recientemente, esto se ha realizado principalmente mediante la migración en vivo de máquinas virtuales. Hoy en día, los contenedores brindan la flexibilidad para manejar muchos entornos de software y tareas en un esquema de virtualización más liviano, proporcionando una alternativa más ágil a las máquinas virtuales para ciertas aplicaciones. El guardar el estado de una aplicación junto con un administrador de contenedores permite la migración en vivo de un contenedor.

En este trabajo, utilizamos la migración en vivo de contenedores y el monitoreo en tiempo real para desarrollar una plataforma de aprovisionamiento de recursos en la nube que permite una mejora en el uso y ejecución de los contenedores. Como resultado, los recursos disponibles se pueden ajustar y distribuir de una manera más eficiente; proporcionando un mejor uso de la nube. Esta investigación explora el uso de técnicas de migración en vivo de contenedores para mejorar el aprovisionamiento de recursos en la nube. Presentamos el diseño y desarrollo de nuestro conjunto de herramientas de migración en vivo, monitoreo y aprovisionamiento de contenedores, así como una evaluación y caracterización del desempeño.

“If you wish to make an apple pie from scratch, you must first invent the universe.”

— *Carl Sagan, Cosmos*

ACKNOWLEDGEMENTS

I would like to thank my colleges at the Laboratory for Applied Remote Sensing, Imaging and Photonics (LARSIP) for their support and help. I’m grateful for the opportunity my professors have given me and thank my family for their unconditional support. Thanks, the Office of Naval Research for supporting this research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

Compute power is not always used to its potential. We want to use modern software technologies to improve system resource utilization, and performance of running workloads. Our work consisted of developing a cloud platform that achieves this by using container live migration. This work expands the research of container migration techniques, real-time system monitoring and resource provisioning policies. In particular, we have developed tools to handle those tasks, performed analysis and optimizations of our toolset and evaluated workloads using machine learning to construct compute resource provisioning polices. We then methodically evaluate the performance of our entire platform and develop resource provision policies. We have tested our system on two commodity servers running multiple virtual machines to simulate a realistic cloud infrastructure. On this virtualized system, we tested our platform by executing a small number of concurrent container workloads to test system scaling up to thousands of container workloads.

This thesis is organized by the three main components of our container live migration and resource providing platform: Container live migration, resource monitoring and finally resource provisioning. On the first chapter we present an introduction to the problem, terminology and solution. On the second chapter we review research literature related to the topics covered by our work. The third chapter discusses our container live migration toolset, migration techniques and optimizations. The fourth chapter discusses our real-time resource monitoring tool and analysis of its performance. On chapter five we illustrate how we analyzed container workloads using machine learning to develop our container placement polices. Chapter five also discusses the testing framework of our live migration container placement policies and their measured performance. The last chapter provides a brief conclusion of our work, findings, and presents directions for future work.

1.1 Containers

A container is an operating system feature in which the kernel allows the existence of multiple isolated user-space instances [1]. Docker defines a container as “a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another” [2]. Containers are a lighter-weight, more agile way of handling virtualization, compared to virtual machines. Rather than spinning up an entire virtual machine with its hypervisor, kernel and system services; a container packages together everything needed to run a piece of software by sharing the host’s kernel. Containers provide a level of flexibility to handle many software environments and tasks [3].

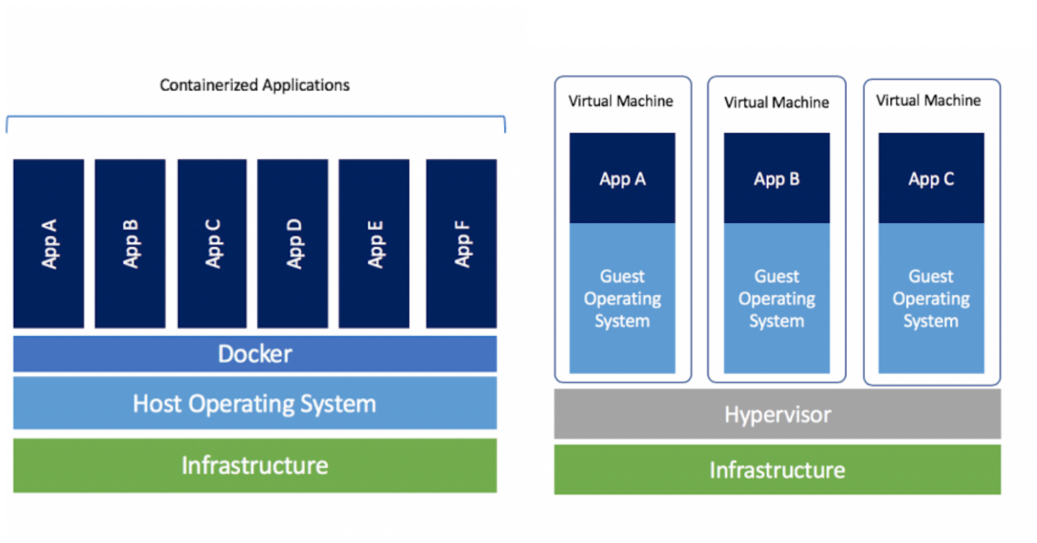

Figure 1-1 Containers vs Virtual Machines

1.2 Live migration and Application Checkpointing

Application checkpointing consists of saving a snapshot of the application's state, so that it can restart from that point. *Checkpoint/Restore in Userspace* (CRIU) is a software tool for the Linux operating system that allows to freeze a running application (or part of it) and checkpoint it as a collection of files on disk [4]. These files can then be used to restore the application and run it exactly as it was during the time of the freeze. The use of a software tool like CRIU for application checkpointing, together with a container manager as Docker, allows a developer to migrate a running container to another host.

1.3 Resource Provisioning

Resource provisioning describes the process of assigning appropriate resources to compute workloads. These tasks require the pairing of the best compute resource to a workload based on its requirements. This is a complex job that needs to be optimized for the best use of the available system resources. We monitor in real-time system resource utilization metrics to determine system bottlenecks. We use container live-migration to improve system performance by evaluating the available resources on the infrastructure to better accommodate a container. Resource provisioning evaluation and container migration decisions are determined by our resource provisioning policies. We developed those policies after performing system and container workload characterization experiments.

1.4 Machine Learning

Machine learning algorithms build a mathematical model of sample data, in order to make predictions or decisions without being explicitly programmed to perform the task. We used machine learning to analyze system resource utilization, it allows us to select the most important features of a given workload to focus our resource provisioning policies on those system metrics. With machine learning we can forecast the performance of a container's workload on another compute host to improve our container live migration policy.

1.5 Problem Statement

Cloud computing has evolved from using virtual machines to containers. Most cloud provisioning services, and software as-a service (SaaS) providers, use containers for their “serverless” services. Current software stacks for managing containers, allow automatic resource allocation by previously selecting host and keeping them in hot-standby. Such case is Kubernetes’ horizontal autoscaler, that uses user-defined metrics, to allocate containers on a set of available hosts using a statistical algorithm. [5]

Current software stacks for container management do not use live migration for performance tuning. Workload managers such as Slurm use application checkpointing for failure migration but not for resource provisioning or resource balancing. [6]

In this work we present the developed of our container live migration platform that sets an ideal configuration of container placement. It manages compute resources for the current running containers based on the available resources so that they are utilized in the most efficient manner, through the use of live migration.

We conducted workload characterization to understand container resource utilization to design a container placement policy for live migrations. We used machine learning for resource feature selection, workload forecasting and develop a methodology to estimate container performance gains with live migration.

This research contributes with the development of a container live migration toolset for Docker. Our toolset optimizes container live migration adds support for network connected containers. We also developed a container state transfer scheme by migration the containers filesystem, in an optimal manner. We present performance analysis of our live-migration techniques. We also contribute with the developed of a lightweight container monitoring tool that supports live migrating containers.

For this research, we used a set of comity machines, servers and IoT devices. Multiple machines act as container hosts and run the live-migration platform. A central machine runs the monitoring software and runs the data processing and provisioning model to perform the resource allocation decisions.

1.6 Container Live Migration System

Our live migration toolset for containers defines Compute nodes and a Control node that together act to manage and execute containers. The Compute nodes hosts the containers and are responsible for orchestrating the container live migration tasks, as well as receiving communication from the Control node. The Compute node will send status and monitoring information to the Control node.

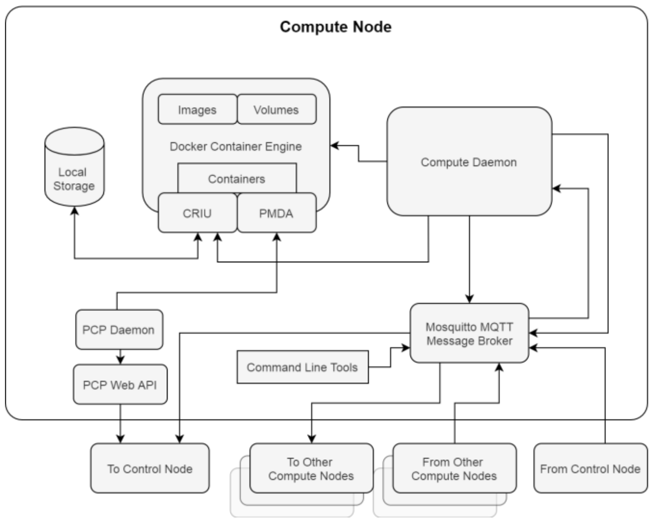

Figure 1-2 Compute Node

The Control node is responsible for monitoring the containers and Compute node's performance by processing real-time monitoring information. It determines the ideal container placement configuration based on current and past resource metrics. The Control node sends container migration request to the Compute nodes.

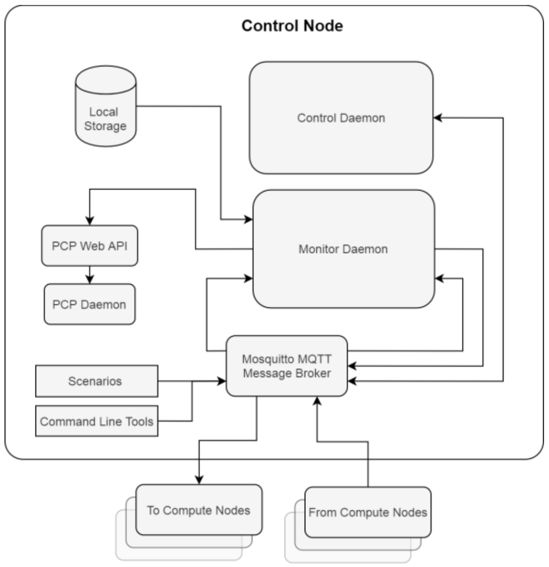

Figure 1-3 Control Node

All messaging is done through message brokers on the Compute and Control nodes. The Compute nodes manage container image caching and container filesystem transfers to increase performance during the live migration process. Part of the toolset is a set of command line programs to run container live migration from Compute host to Compute host. These tools were used to test our platform components and perform system characterization and resource provisioning experiments. We name our platform Herd, as it manages a herd of containers (Figure 1-4).

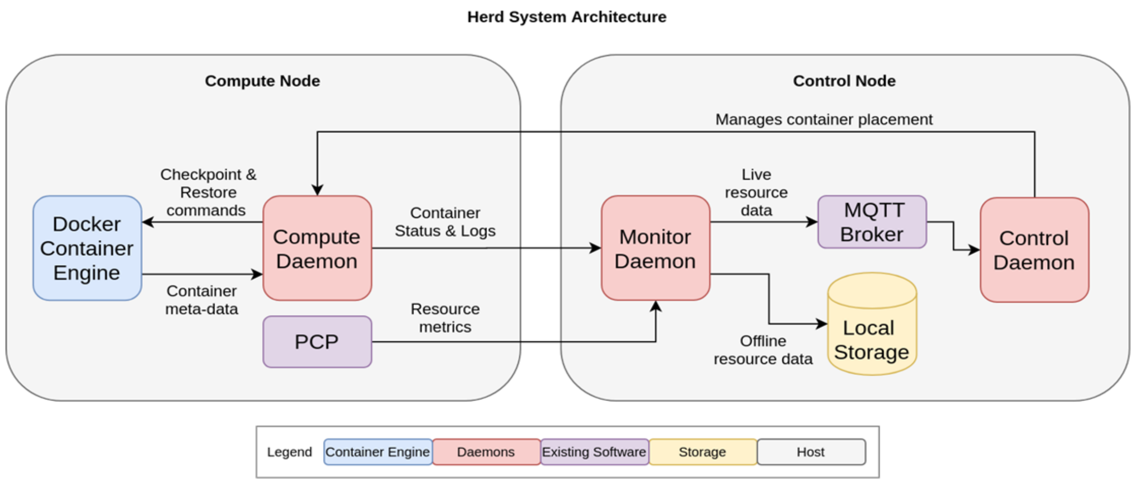

Figure 1-4 Herd System Architecture

2 LITERATURE REVIEW

Performance analysis of container live migration was the main research point by Berg et al. [7] on their work with container checkpointing for distributed system. Their work showed that although failure is common, it is possible to live migrate containers. Kudinova et al. [8] developed a mathematical model for handling process tree in containers for live migrations, as a container does not have a main "init" process for task management. The live-migration sequence requires the process tree inside the container to be restored in a specific way.

For fault tolerance, Lee et al. [9] presented a platform that uses a container-based light virtualization. Their platform uses an automated build function to isolate an application so that live migration function can be used. They perform this isolation operation to ensure that systems have a high reliability in cases of hardware failures. Merino et al. [10] proposed a resiliency technology to fight through cyberattacks. In particular, they designed a platform to orchestrate and manage the container lifecycle while enforcing security and applying resilient techniques. Their platform allows to deploy an application, enforce its security, and return it to a secure state in case of a cyber-attack. They attempt to achieve these goals with techniques such as live migration, checkpointing and container cloning.

For resource management and system load management, Arzuaga and Kaeli [11] worked on virtualized servers and virtual machine live migration. Their work yielded the development of a statistical algorithm to quantity server load imbalance. We used the concepts behind this algorithm to develop our container placement policies

Rolim's et al. [12] work on machine-learning algorithms and fuzzy logic for resource management did not include the use of container or live-migration. Their work served as a guide on what algorithms and metrics to explore.

Biswas et al. [13] presented a technique for auto-scaling of resources that dynamically changes the number of resources for the private a cloud based on system load. Their technique supports on-demand and advance reservation requests by using machine learning to predict future workload based on past workloads. Their results demonstrate that these techniques can effectively lead to a reduction of overall cost and usage of cloud resources.

For resource provisioning and task scheduling for cloud service providers, Cheng et al. [14] worked on a deep learning model designed to automatically generate the best long-term decisions. Their model learns from the changing environment to determine the price for a cloud service. They used artificial intelligence techniques such as target network, experience replay, and exploration and exploitation to develop a model that provides a high energy cost efficiency.

Cortez et al. [15] worked on workload forecasting using machine learning to improved resource management. They developed a system called Resource Central that gathers virtual machine performance metrics for analysis. They used gradient boosting to predict resource utilization of a virtual machines to better manage horizontal scaling of cloud infrastructure. They also researched on cyclical workload characterization to improve their machine learning model for resource provisioning in a cloud environment.

Ahmed et al. [16] worked on using checkpoint and restore with docker containers to optimize the deployment of distributed fog infrastructures. They improved loading time of services on low resource edge-computing devices. They used application checkpoint to preload and cache the services deployed on their infrastructure.

Wood et al. [17] research on resource management using virtual machine live migration, Sandpiper, presented a system that automates the tasks of monitoring virtual machines resource utilization. Their research presents the obstacles they had to handle for virtual machine monitoring for live migration and the mitigations they used to avoid them. Their platform sets an evaluation period to characterize a virtual machines workload and define a resource utilization baseline. With this baseline their Sandpiper system can automatically migrate virtual machines for better resource provisioning.

Mergenci et al. [18] worked on a generic resource allocation metrics and method for heterogenous cloud infrastructures. They propose a multi-dimensional best-fit algorithm that handles multiple resource metrics to compute an ideal compute host and virtual machine combination. Their algorithm improves resource utilization of their infrastructure.

Very recently, Souza et al. [19] worked on stateful container migration in geo-distributed environments. They used application checkpointing to migrate containers around large-scale geo distributed fog-computing infrastructures. They demonstrated that using container filesystem migration improves application performance.

3 CONTAINER LIVE MIGRATION

3.1 Containers

Containers provide process isolation by the use of Linux kernel features. Containers can be daemon-based or daemon-less. A daemon-based container, such as Docker, is managed by a daemon process that handles the container's lifecycle. Daemon-less containers, such as Podman, directly use kernel features through a runtime [20]. A container runs with a base image, this image holds the containers filesystem. "Docker makes use of kernel namespaces to provide the isolated workspace called the container. When a container is executed, Docker creates a set of namespaces for that container. These namespaces provide a layer of isolation. Each aspect of a container runs in a separate namespace and its access is limited to that namespace. […] Docker also makes use of kernel control groups (CGROUPS) for resource allocation and isolation. A CGROUP limits an application to a specific set of resources. Control groups allow Docker Engine to share available hardware resources to containers and optionally enforce limits and constraints." [21] A container is a process that is isolated from others through the use of namespaces and resource isolated through the use of control groups.

3.2 Application Checkpointing and Live Migration

Application checkpoint is process of freezing an applications state as a collection of files. The application can be restored from these files. CRIU is a software tool that provides these features. A checkpoint contains a process file descript information, memory maps and other stateful objects such as TCP sockets. Application checkpoint is a complex process, not everything can be checkpointed by CRIU. File locks, devices, and task from other users are among the things that cannot be checkpointed by CRIU [22].

As a container is a process, it can be checkpointed. Docker provides, as an experimental feature, the ability to checkpoint and restore a container. Container checkpointing does not handle the container's filesystem. Fundamentally, one form of implementing live migration would be by transferring a checkpoint to another machine. Docker itself does not feature container live migration. However, our approach to live migration consists of checkpointing a Docker container's state, transferring it to another machine, and restoring said state on a newly deployed container.

As mentioned, our container live migration scheme is achieved by performing a series of tasks in the most transparent manner. We define two Compute nodes, one as a source node and the other as a target node. We perform container live migration from the source to the target. Docker checkpoint and Docker restore are orchestrated to recreate a new container in the target host with the context of the container from the source host. The transfer of the checkpoint files are automated and once it is complete, the container restore operation is performed. Figure 3-1 illustrates all of the required operations.

The checkpoint and restore (C/R) tasks for container migration (Figure 3-1) include:

1. Send CRIU Checkpoint command to Container A in Source Node
2. Checkpoint Container A
3. Create Container B in Target Node
4. Transfer Checkpoint A to Target Node
5. Send CRIU Restore command to Container B in Target Node using Container A Checkpoint
6. Restore Checkpoint A on Container B
7. Remove Container A in Source Node

The source container is inactive after a checkpoint, the target container is inactive until its restored. A container should not be at two places at once while active. These tasks must be performed as quickly as possible as to minimize container down time during migration.

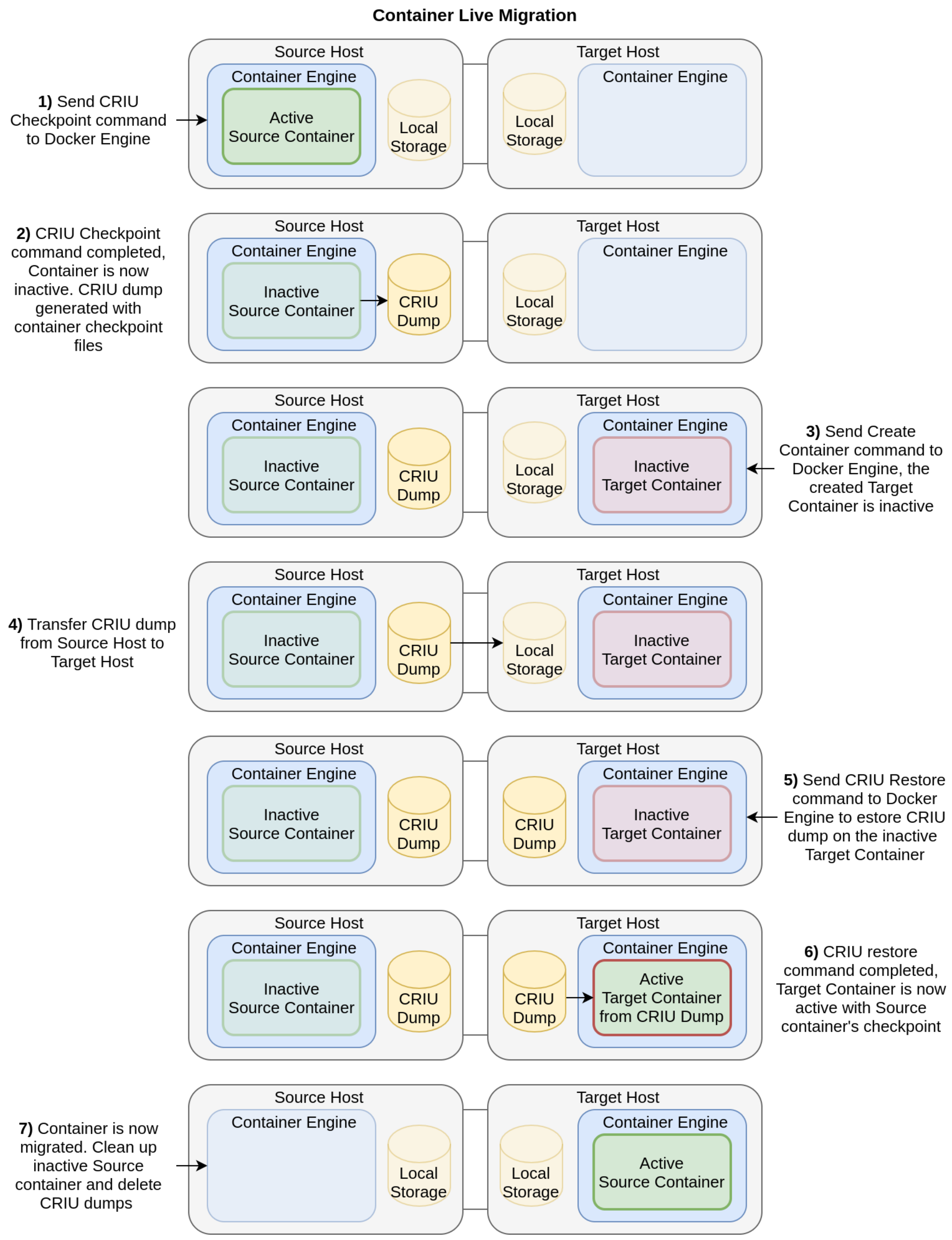
Container Live Migration
1) Send CRIU Checkpoint command to Docker Engine
Source Host
Container Engine
Active Source Container
Local Storage
Target Host
Container Engine
Local Storage
2) CRIU Checkpoint command completed, Container is now inactive. CRIU dump generated with container checkpoint files
Inactive Source Container
CRIU Dump
3) Send Create Container command to Docker Engine, the created Target Container is inactive
Inactive Target Container
4) Transfer CRIU dump from Source Host to Target Host
5) Send CRIU Restore command to Docker Engine to estore CRIU dump on the inactive Target Container
6) CRIU restore command completed, Target Container is now active with Source container's checkpoint
Active Target Container from CRIU Dump
7) Container is now migrated. Clean up inactive Source container and delete CRIU dumps
Active Source Container

Figure 3-1 Container Live migration checkpoint and restore sequence

3.3 Herd Compute Overview

3.3.1 Compute Daemon

In each Compute node of our container resource provisioning platform, runs a compute daemon that handles the sequence of tasks required to manage container live migration. This compute daemon receives migration requests from the Control node or manually from a user. We developed the container compute daemon, called Herd Compute, that takes care of performing automatic container live migration. Herd Compute also handles container filesystem migration and container network configuration from a source Compute host to a target Compute host. It keeps track of the container state over different compute nodes.

3.3.2 Container Migration

The compute daemon migration by issuing checkpoint and restore commands to the Docker Container engine on the compute nodes that are acting as source host and target host. The daemon handles communication with other hosts using bidirectional communication over MQTT [23].

3.3.3 Container output logging support for migrations

The container engine, Docker, keeps a log of the standard output of a container. During a container checkpoint this log is not exported, and the log is lost when a container is migrated as the source container is destroyed. Herd Compute parses the container log in real time and streams it over MQTT to be read by other clients, such as the monitor daemon. We implemented this feature initially for debugging purposes but later it served to validate a container workload's performance.

3.3.4 Container Networking

Herd Compute handles live-migration of network-connected containers by rebuilding the network configuration on the target host from the information present on the source host.

This works without any user interaction, as the network information is obtained from each container's configuration.

3.3.5 Container Filesytem

Herd Compute can handle container filesystem migration as a configurable option. We have tested different filesystem migration techniques and optimizations and settled for the overlay filesystem layer migration because it offered the best performance.

3.3.6 Healthcheck

We implement a watchdog daemon to make sure the compute daemon is always available and can process requests from the Control node. Herd Compute publishes its status information so the system can acknowledge its availability. To handle live migrations, the compute daemon inspects the container, to make sure no errors may have occurred. It reports its findings to the client that issue the migration command.

3.3.7 Herd Compute architecture

Herd Compute is a multi-threaded software written in Java, its purpose is to orchestrate between a source Compute node and target Compute node the task to achieve container live migration (Figure 3-2); its main components are:

- Daemon thread: is the main thread of the program that starts all other threads.
- Migrate thread: orchestrates the live migration tasks by executing other threads and sending messages to the target host so it executes the required tasks.
- Checkpoint thread: handles the container checkpoint tasks, on the source host, by communication with the Docker container engine.
- Create thread: handles the container create tasks, on the target host, by communication with the Docker container engine with information provided from the source host.

- Transfer thread: handles the checkpoint and filesystem transfer to target host tasks, on the source host.
- Restore thread: handles the container restore tasks, on the target host, by communication with the Docker container engine and restoring the transferred checkpoint on the container created by the create thread. It also handles filesytem restoration.
- Inspector thread: gathers container information and checks if a migration was successful. It also provides the ability for remote clients, like the Control node, to obtain container information before or after migration.
- Rollback thread: handles the cleanup task, on the target hosts, in case a step of a live migration fails. It is initiated by the source host.
- Logger thread: parses the container standard output log and publishes over MQTT, it is initiated by a remote client request such as the monitor daemon.

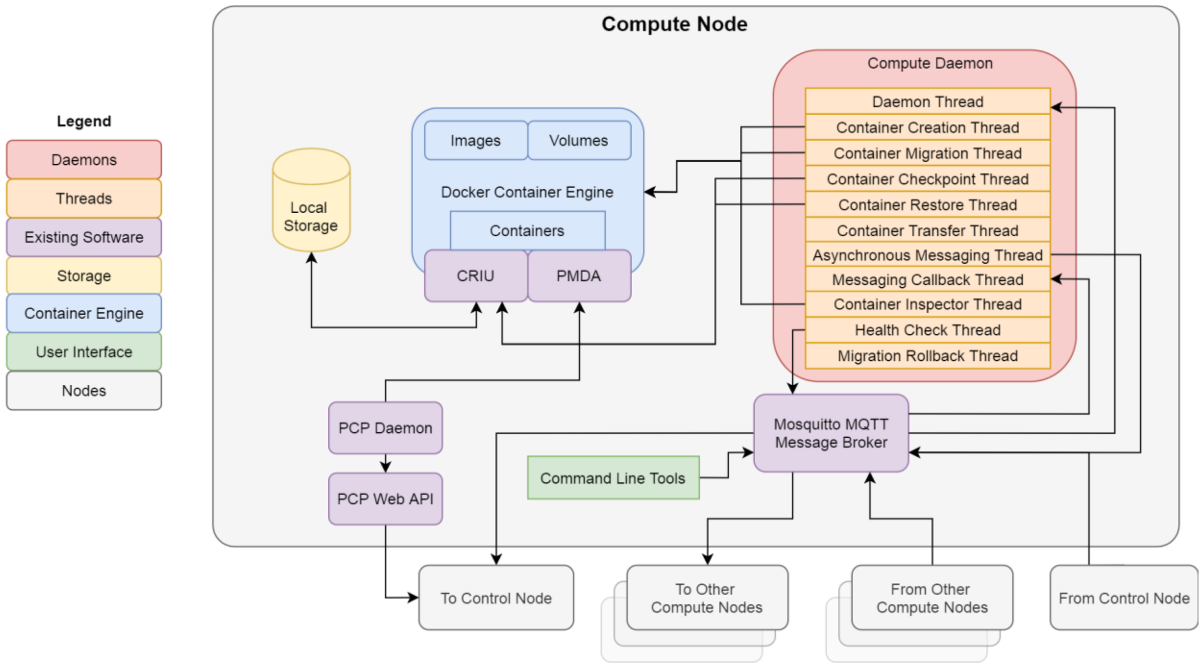

Figure 3-2 Detailed Herd Compute with its threads

3.3.8 System Messaging

Herd Compute manages each request independently from other requests, by launching a thread per request. This allows a Compute host that is currently acting as a target host and receiving a container to also act as a source host and also be sending a container. The multi-threading features of Herd Compute allows the Herd system to asynchronically to handle concurrent container migrations. It also allows Herd to handle other request such as status reporting and container health check at the same time.

All messaging on the system is done asynchronously and bidirectionally using the publish-subscribe MQTT protocol. Each Compute node runs an MQTT message broker that is used to communicate with other nodes. When a node needs to communicate, it publishes messages on the other node's broker (Figure 3-3). Each node is subscribed to its local broker. The Compute nodes orchestrate themselves the task required to live migrate a container without the need of a system-wide (multi-hosts) centralized entity.

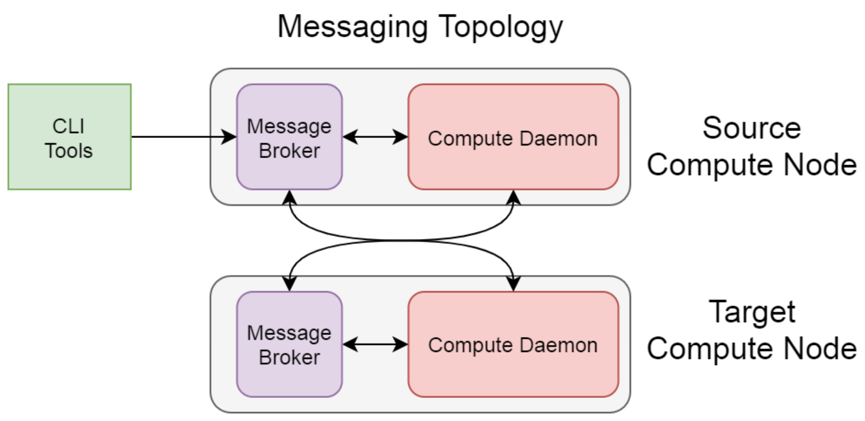

Figure 3-3 Compute Daemon Messaging topology for container live migration

3.4 Optimizing live migration

Through the development of Herd Compute, we tested and implemented different container live migration techniques. Our initial prototype technique was a scripted checkpoint-only stateful migration. It only handled container state but did not handle filesystem or network. This was a serial live migration with no optimizations where every stage of the process is executed one after the other.

After the initial serial container live migration test and prototypes, we developed an optimized live migration sequence (Figure 3-4). It parallelizes the task of creating the target container on the target host and the checkpointing the container on the source host. This sequence was the first implemented and tested in Herd Compute. The design and development of the Herd Compute daemon revolved around parallelizing tasks by running asynchronous threads. This parallel container live migration sequence proved to be faster than the initial serial one.

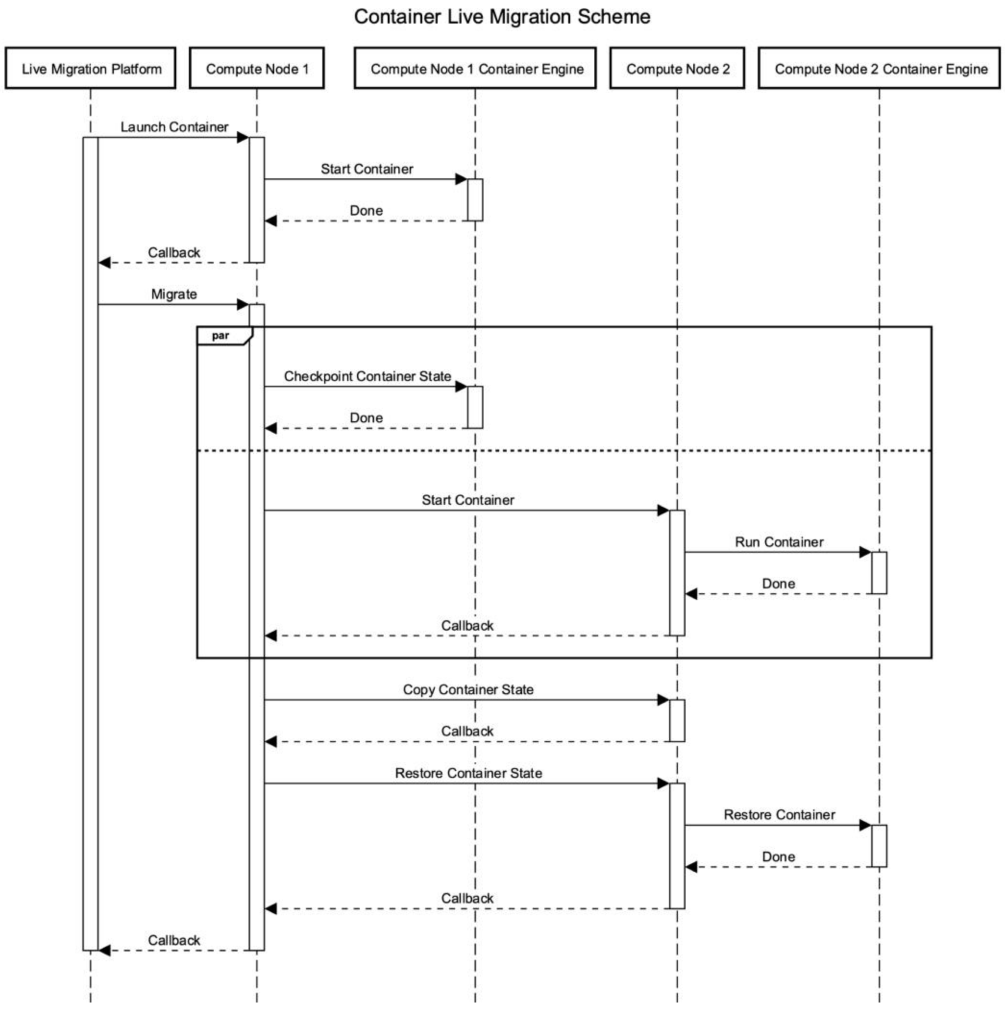

Figure 3-4 Asynchronous live migration sequence diagram

To test live migration, we implemented a series of programs that communicate with Herd Compute to initiate a container live migration. The first is a command line application, we named Herd CLI. It takes as parameters the container source host and target host's name or IP address, and the container name or ID to be migrated from the source host to the target host. Herd CLI supports sending the compute daemon migrations options such as filesystem transfer mode or compression during transfer. We used this application extensively to test live migration techniques, system deployment, CPU architecture compatibility among other tests.

To automate our test, we developed another live migration application named Herd Pong. It automatically migrates a given container from a given host to a series of hosts defined on a configuration file. We used this to test sequential migrations from one host to another and random migrations to any host for stress testing and concurrency management testing. Herd Pong keeps a container on a given host for a given amount of time and migrates it a given count. It was used it extensively for workload characterization and is the main tool behind our "staircase" methodology explained in chapter five.

3.4.1 Concurrency management and Stress testing

The multi-threaded design of Herd Compute manages concurrency for its own shared objects. During our tests we found that CRIU does not handle well concurrent requests. By issuing a checkpoint command to Docker, CRIU locks its own resources and a concurrent checkpoint or restore commands will fail, if not handled property by Herd Compute. To test these effects, we developed concurrency and stress tests. These tests concurrently migrate containers, compute hosts send and receive containers at the same time, without any delay between migrations. Some container migrations failed and CRIU logged resource access errors. To mitigate such errors, we synchronized Herd Computer's checkpoint and restore threads at the point where each thread issues commands to the Docker engine. The results of these optimization were faster live migrations and no resource access errors by CRIU (Table 3-1).

	Non-Synchronized CRIU	**Synchronized CRIU**
Migration Time (seconds)	1.82s	1.76s

Table 3-1 Migration stress testing results after concurrency

3.5 Container networking

3.5.1 Introduction

For network-bound container live migration like those in Online Transactional Processing (OLTP) [24] workloads, we explored multiple alternatives that would allow us the automatic reconnection of a container on a new host. CRIU supports checkpoint and restore of TPC and UPD sockets, so live migration of a network bound container should be possible in most cases. The problem arises in the management of IP address, hardware MAC address and traffic redirection when the container moves from one compute host to another.

Red Hat’s work on container live migration [25] used a High Availability (HA) and IP-failover management software called KeepAlivedD. It uses Virtual Router Redundancy Protocol (VRRP) to define a floating IP for multiple hosts [9], given a correct configuration between them and the setup of host priority. VRRP maps a virtual network adapter with a special MAC address to the floating IP that KeepAlivedD will use to redirect traffic from one host to the other in case of failure. VRRP is implemented directly into the TCP/IP protocol and will automatically route the traffic from the defined mac address mapped from the floating IP. KeepAlivedD will issue the MAC address change/redirection command when it detects that a host is not available. After exploring KeepAlivedD and Red Hat’s previous work, we found that their solution was too limited for our platform.

Docker container networking works based on *Docker network drivers*; the default container network configuration is the *bridge* mode. In this network mode the container is connected to the host computer using a network bridge. “A bridge network uses a software bridge which allows containers connected to the same bridge network to communicate, while providing isolation from containers which are not connected to that bridge network. The Docker bridge driver automatically installs rules in the host machine so that containers on different bridge networks cannot communicate directly with each other.” [26] This default configuration is set as it allows container network isolation from the host while keeping the ability of containers to communicate with each other if using the same bridge. To connect to a container from another machine, each container network *port* that needs to be accessed must be *exposed* from the container in its host machine. Bridge mode works on a single machine and does not provide us with a viable way to handle live migration containers.

Docker supports *overlay* networks that connect multiple Docker daemons together by setting up a network driver that creates a distributed network among multiple Docker daemon hosts [27]. Overlay mode works with Docker Swarm [27] and other service discovery frameworks like Consul [28]. Although this could potentially work, the added complexity of setting up a swarm service to enable network-bound container live migration pushed us to keep exploring for other alternatives.

Docker *Host* networking is another network driver that completely exposes the container network to the host [29]. It removes all network CGROUP isolation from the container and merges it with the host CGROUP networking. This is the network mode that Red Hat used, it allows the use of KeepAliveD by setting up the failover IPs as the compute host IPs. We tested this networking mode by setting up two Compute nodes and a web server running inside a container on each host and a floating IP for both nodes. We simulated a migration by disconnecting the compute host from the network and monitoring the network traffic with Wireshark. KeepAlivedD detected the disconnection based on its *keep alive* timer and sent the VRRP redirection command to direct traffic to the other host. In our case, this mode is problematic as it merges container and host networking and will only allow one service to bound a host port, making it work only for one container per Compute host per workload.

To mitigate the limitations of Docker *Host* networking we explored the idea of using MACVTAP network adapters. We could bridge to the host network and reconfigure

Docker *host* networking to the MACVTAP pair as its main network adapter. This setup is used for network-enabled KVM/QEMU virtual machines [30]. MACVTAP is a device driver that simplifies virtualized bridged networking. When a MACVTAP instance is created on top of a physical interface, the kernel also creates a character device TAP which can be directly used by KVM/QEMU. With MACVTAP, you can replace the combination of TAP and bridge drivers with a single module [28]. This setup will need us to manage a MACVTAP pair for each container on each host, and would potentially need to reconfigure KeepAlivedD in real-time every time a migration is conducted; as we need to define the target host as a failover IP; and also to recycle IP address, as VRRP can only support a handful at a time [31]. After more Linux and Docker network research we found the MACVLAN adapter.

3.5.2 MACVLAN Adapter

With MACVLAN, multiple virtual network interfaces with different unique MAC addresses can be created on top of a single physical network adapter. Without MACVLAN, to connect to physical network from a VM or namespace, a TAP/VETH device needs to be created (Figure 3-5). One side of the VETH device attached to a VM, the other side connected to a bridge and the bridge connected to a physical network interface on the host at the same time [32].

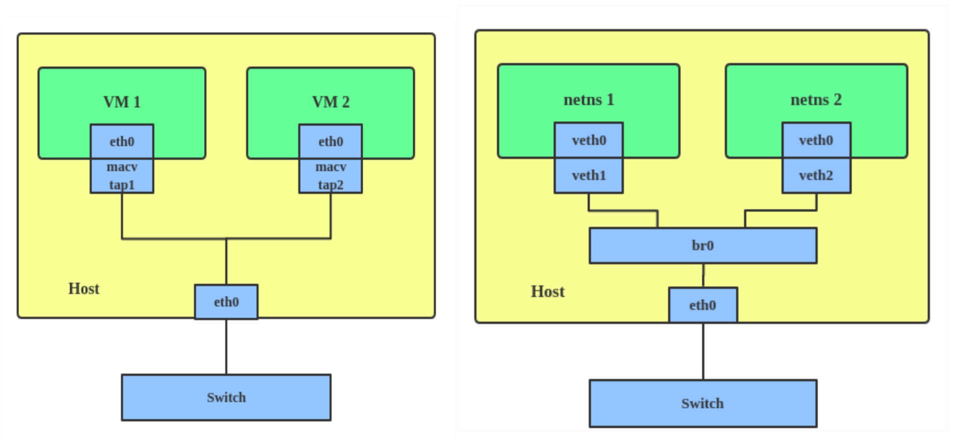

Figure 3-5 MACVTAB and TAP/VETH network adapters

There are multiple MACVLAN modes, we settled with *Bridge* mode. It allows all endpoints to be directly connected to each other with a simple bridge via the physical interface (Figure 3-6). Containers on the same host are be able to communicate with each other directly through the host network interface without having to reach the network router. The container compute host is no able to access the container through its own network interface when using a MACVLAN adapter. We understand that for our case, this is acceptable as in most cases containers will not be accessed from the Compute host but rather the Control node. Containers are accessed from any other machine that isn't a Compute node as containers will be migrating from Compute host to another. This limitation can be mitigated by setting up another MACVLAN adapter in the Compute host and defining a route to the container's subnet. We used this approach during our development of Herd Compute.

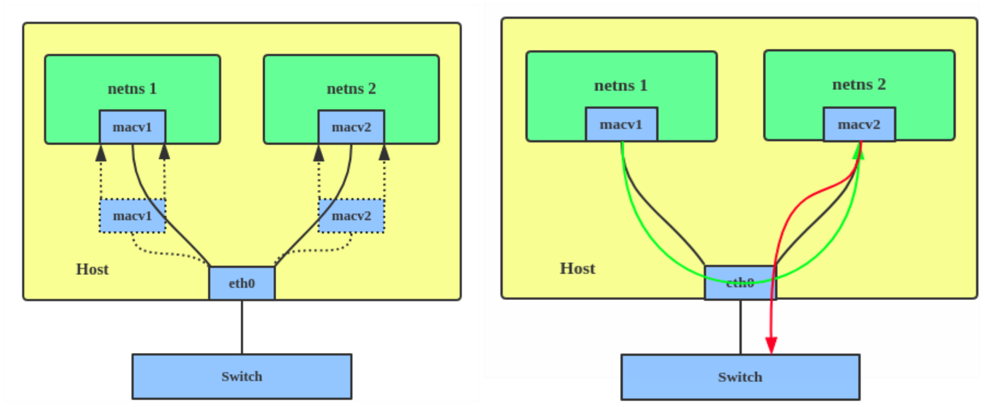

Figure 3-6 MACVLAN adapter and MACVLAN adapter acting in bridge mode

The MACVLAN adapter provides us with the ability to assign each container its own MAC address and IP address. This allows us to restore a container while keeping its original address, without the need of setting up a floating IP or using KeepAliveD. The MACVLAN adapter is a device that works at the *data link* layer of the OSI network model. As such, the traffic at *network* layer only suffers an outage and TCP/IP takes care of handling dropped packages during live migration.

Docker supports MACVLAN device as a *network driver* to assign a MAC address to each container's virtual network interface. This allows the container's network interface to be directly connected to the physical network. To do this, a Docker MACVLAN network needs to be created designating a physical interface on the Compute host as the parent of the MACVLAN device. This Docker network need to be configured with the subnet and gateway of the physical network. At the container creation moment, we define the container's network *driver* as the created Docker MACVLAN network, and we assign an IP outside the DHCP range of the physical network.

We tested container networking using MACVLAN by setting up a Docker network with the same gateway and subnet as the physical network. We assigned the Docket network parent adapter as the network adapter connected to the physical network on the Compute host. We deployed two containers and assigned them an IP not in use on the physical network. We tested basic network connectivity between two containers using tools like *PING*, *traceroute*, *nmap* and *iperf3*. We monitored network traffic on the host with Wireshark by assigning a MACVLAN interface connected to the physical adapter and setting up individual routes to each container. The containers were running in privileged mode and with network admin mode (CAP_NET_ADMIN) allowing us to test and monitor the virtual network device within the container. After the initial successful test, we ran the containers without network admin mode and in non-privileged mode and had the same results.

To live migrate a container with network support, the target container must be recreated using the same network configuration as the source container. The source container is automatically inspected by Herd Compute to then send the *create* command with the respective network parameters to the target host. The Herd Compute daemon on the target host will create the container with the correct network configuration. These parameters include network name, network mode, network subnet, network gateway and container IP address.

To test network-bound container live migration, we implemented changes to Herd Compute and tested a live migration of a container while connected over SSH. We had successful network communication reestablishment between containers after live migration. To test network-bound container live migration of two containers connected to each other that also needed filesystem migration support, we deployed a new testbed on top of an Openstack hypervisor with three Compute nodes. We deployed an OLTP test workload by setting up an OLTP server and two OLTP clients. We achieved successful implementation of automatic network-bound container live migration changes to Herd Compute.

3.6 Filesystem migration

Checkpointing a Docker container does not migrate the container filesystem [33]. This causes problems when migrating containers running workloads that require disk access such as OLTP workloads. To test container live migration with filesystem support, we developed a scripted prototype that exports the whole container as a TAR file. Exporting the whole container using the Docker commit command flattens the container filesystem changes to the base image as well as the base image itself. The final exported file takes as much space as the image and the changes (file additions and even previous subtracted files). This TAR file is then transferred to the target host, imported to Docker to then create a container based on this imported image as a base image. The checkpoint is then restored on the created target container.

This whole container filesystem migration process can take up minutes as the source container needs to be flattened, exported, transferred, imported, deployed and checkpointed in serial as each step depends on the previous step. The image transferred is big, taking much of the network transfer speed. The export and import process is slow as the container engine needs to save every file on both the container and base image. This whole container filesystem migration works, and the restored container has all the files and continues executing without issues. The live migration is extremely slow, but this is initial proof of concept to the possible benefits of filesystem migration.

Containers work using an overlay filesystem (OverlayFS). The overlay filesystem is a difference-based layer filesystem where each layer is a change to the previous layer. It consists of a *lowerdir* layer that acts as a base, and in the case of containers is the base image layer(s). The *upperdir* is the copy-on-write (COW) changes made by the container to the filesystem. This *upperdir* layer is the container layer from the container engine's perspective. The *lowerdir* and *upperdir* layers are joined to a *merged* layer, this layer is the filesytem the container can access (Figure 3-7). All compute hosts have a copy of base container images we used for testing and those used by the workloads. The compute engine is capable of downloading a missing base image from the online image repository Docker Hub. We exploited the concepts behind the OverlayFS structure for our benefit by migrating only the *upperdir* layer during live migrations.

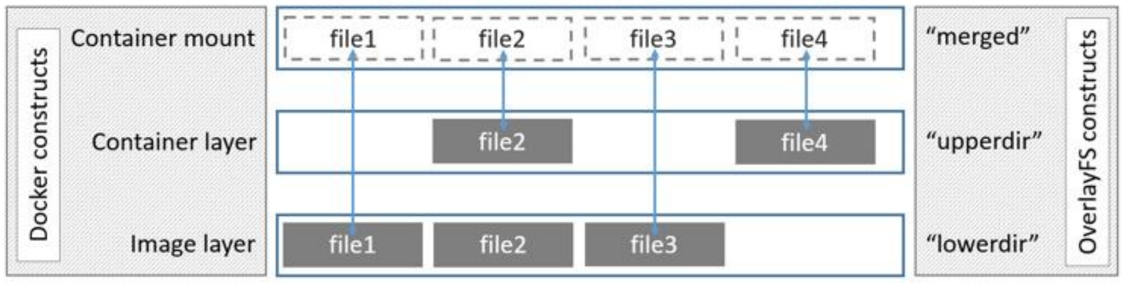

Figure 3-7 Container filesystem structure using Overlay2FS

To optimize our container filesystem migration scheme, we want to migrate only the changes made by the container to the base image. As a container's filesystem works by using copy-on-write, only the changes to the base image are stored. The *upperdir* layer contains by itself all the changes and thus we don't need to compute the difference to the base image. To achieve this, we need to know the container filesystem ID and location, which differs from the container ID and CGROUP ID. We tested our hypotheses by developing a migration script that obtained the source container filesystem location on the Compute host using Docker *inspect* command. It moves the *upperdir* folder to the target hosts, copies it to the *upperdir* layer of the target container's *upperdir* location and restores the checkpoint. Our initial tests were successful and proved that by only migrating the container's *upperdir* layer, a quicker container filesystem migration was possible. We named this method difference layer migration or Diff layer for short.

To implement the new FS migration scheme on Herd Compute we used the Inspector thread on the compute daemon to obtain the container filesystem location information from the Docker *inspect* command. We successfully implemented our container filesystem Diff layer live migration changes to Herd Compute improving filesystem transfer by up to 10x in some cases (Table 3-2).

Filesystem Migration mode	OLTP Client-container Migration duration (seconds)	OLTP Server-container Migration duration (seconds)
No FS transfer	3.06s	2.49s
Whole container FS	22.68s	26.89s
Diff layer	2.58s	2.69s

Table 3-2 Filesystem migration duration comparison

3.6.1 Improving difference layer container filesystem migration scheme

We tested transferring the Diff layer and container checkpoint dump files using compression, by transferring the files as a compressed TAR file using *bzsip*. We also tested transferring the files non-compressed but rather using compressed transfer with *SCP*. In both cases filesystem live migration ended up being slower than non-compressed as the limiting factor was the compression time. We tested different compression ratios, but none proved to be faster than non-compressed transfer. The compression features are present as migration options in Herd Control as they could potentially be used for Edge/IoT devices with low network transfer speed.

Because each Compute host has a copy of the base image used by all workload containers, a deployed OLTP server Diff layer will not drastically change unless an OLTP client connects to the OLTP server. That's why our previous tests show little transfer time difference between a an OLTP client or an OLTP server. To tackle this, we tested container live migration of an OLTP server during a client connection and at idle. A connected OLTP server container live migration fails without a filesystem transfer, as the sever has opened files (Table 3-3).

Filesystem Migration mode	**OLTP Server-container Migration duration (seconds)**	**OLTP Server-container with a connected OLTP Client Migration duration (seconds)**
No FS transfer	10.77s	Failure
Compressed Diff layer + Checkpoint (bz2)	73.37	113.21s
Diff layer	19.146s	24.93s
Compress Diff layer (bz2)	54.434s	113.91s

Table 3-3 Compressed Filesystem migration duration comparison

Our initial implementation of Diff layer filesystem transfer copied the layer from the source host to a temporary location, then transferred it to temporary location at the target container and finally copied it to the correct system location at the target host. We named this transfer technique *Staged Diff layer* transfer. It was implemented this way to transfer both the container checkpoint and the filesystem because at that moment in the migration sequence the target container filesystem's location it is not known by the source compute host. The target container has not been created, or the create-callback by the target compute host has not been received by the source host.

To improve transfer efficiency, we tested transferring the container and its Diff layer asynchronously from the container's *create* and *restore* command in Herd Compute. The Diff layer is transferred immediately after the container checkpoint is finished. We named this transfer technique *Direct Diff layer* transfer. The Herd Compute *create* and *transfer* threads are then joined to send the *restore* command. The container *restore* thread at the target compute node takes care of moving the Diff layer to the correct location to then restore the checkpoint. This increased live migration speed, because the transfer is asynchronous from the container creation, if the *create* thread fails in the target host, a callback is sent to the source host to stop the transfer, abort the migration and remove the checkpoint files (Table 3-4). A migration *rollback* command is sent to the target host to remove the created container and transferred files (as even in error, a dead container may be created).

Filesystem Migration mode	**10 Warehouse TPCC Server-container with a connected client Migration duration (minutes)**
Staged Diff Layer: Rsync no compression, staged transfer	4.15m
Direct Diff Layer: Rsync no compression, direct transfer	1.32m

Table 3-4 Filesystem transfer mode duration comparison

Network bound container live migration is handled automatically by Herd Compute and does not require special options, container Diff layer transfer option supports network bound containers as they are handled by different threads (*create* thread vs *restore* thread).

Docker's container filesystem does not include special files, like the *Hosts file*, that maps hosts-names to IP addresses. Each container has a unique host name and Docker dynamically generates a *Hosts file* per container. Our tests show that live migrating the filesystem, and even rewriting the source container's *Hosts file* directly on the target's containers filesystem, does not work. This causes errors when testing network bound container live migration as every time a container is migrated, its *hostname* changes, but its restored state contains the previous *hostname*. A *hostname* and *Hosts file* mismatch may cause latency errors when running workloads within the container. To mitigate this, Herd Compute sets the original *hostname* of a live migrated container, by using Docker command line parameters. This original source container *hostname*, that we call *PrimalID* is saved on each Compute host and copied to any target host during filesystem transfer.

3.7 Migration rollback

In the case of migration error, the Herd Compute can rollback a migration by using the same checkpoint and restore features used for migration. During our initial tests we ran the container checkpoint command using the “--leave-running” Docker option so CRIU does not terminate the container at the moment the checkpoint is created. When not using that option, if the container checkpoint is successful, the container will be stopped. After many tests we found out that the container will stay running in the case of a checkpoint error and will stop in the case of a successful checkpoint. This was used for our benefit to handle migrations errors.

Herd Compute is designed so that if a container checkpoint fails on a source host, the checkpoint files will not be transferred, and a *rollback* command will be sent to the target host. This removes the target container that was created and is waiting to be restored. If a *container create* error occurs at the target host, the compute daemon on said host will send an error status in its callback to the source host and the source compute daemon will stop the transfer and handle the error by sending a *rollback* command to the target host to delete the inactive target container. If a container fails to be restored, the target host’s compute daemon will send an error status in its callback to the source host and it will handle the error by sending the *rollback* command to the target host.

Herd Compute handles errors in three main ways. If checkpoint fails, the compute daemon rollbacks the target container and keeps the source container running. If remote restore fails (on a target host), the compute daemon rollbacks the target container and restores the source container from the checkpoint. If local restore fails, the compute daemon rollbacks the target container and restarts the source container.

Each case is handled by the Herd Compute *Inspector* thread that checks the status of a container to verify if the restore or restart was successful. It is possible for a container to be successfully restored, but its context damaged because of an improper checkpoint or restore, due to CRIU errors. In those cases, once the container is restored its execution may instantly end, so the container status should not be based only on the restore exit status, but rather Docker running status.

A restarted container loses its context and can be considered a failed migration. This error was commonly handled when the system was tested without filesystem transfer support, but after integrating those features, that error is rare. There is also the rare possibility that a *container restart* can fail. This failure can be due to other errors such as, no available drive space, CRIU errors or logic errors. In the case of such error the compute daemon will notify the migration initiator, Herd Control, Herd CLI, or Herd Pong about its status.

4 SYSTEM MONITORING

4.1 Introduction

For our platform we developed a customizable and lightweight real-time resource monitoring software called Herd Monitor. It streams in real-time resource utilization metrics of Compute nodes, its containers and well as performance metrics of workloads running within live migrating containers. For that, the system must keep track of each container and its current hosts, must not impact the resources on the compute host and ideally be independent of the workload within the container. To analyze and validate the metrics, the system must centralize the standard output of the migrating containers, store the data for offline use and provide streaming data for real-time visualization or analysis.

In order to keep the monitoring software as lightweight as possible, existing resource metrics monitoring tools were used. Herd monitor handle all tasks required for automatic monitoring of live migrating containers as well as its current compute hosts.

The initial monitored data is analyzed to identify possible overhead caused by the monitoring system. A baseline was defined to properly fine-tune the system to allow the containers to utilize the available computing resources efficiently. We used offline data generated by Herd Monitor for workload characterization, infrastructure performance tuning and forecasting.

For performance monitoring of cloud infrastructure, there are many software solutions available. Prometheus is a popular solution for container monitoring for its direct integration with Kubernetes [34]. Prometheus can monitor performance metrics from hosts and containers and store them on a time series database. Netflix Vector provides monitoring and real-time visualization by connecting with Performance Co-Pilot to stream and display performance metrics [35]. For visualization Grafana can connect to

Prometheus database [36] as well as Performance Co-Pilot (PCP) to present in real-time performance metrics [37]. We also explored the Docker Streaming API but found that it was too limiting for our needs. Google’s cAdvisor can monitor in real-time Compute hosts and container and export the data over a REST API server. cAdvisor has limitations regarding the metrics it can monitor. Performance Co-Pilot is highly customizable and satisfies all our monitoring needs.

4.1.1 Performance Co-pilot

After exploring multiple monitoring tools and techniques we opted to gather the resource utilization metrics for the Compute nodes and the containers running in them with Performance Co-pilot software (PCP). PCP was chosen over other monitoring software such as cAdvisor as it enables the selection of what metric to monitor. PCP works by running a Performance Metrics Domain Agent (PMDA) that collects data and reports it to the Performance Metrics Collection Daemon (PMCD). This data can then be stored to binary logs or queried by the other PCP tools such as its Web REST API (PMWEBD). Data provided by PCP can be exported to time-series databases or streamed in real-time using its own PCP network protocol.

PCP’s gathers most of the resource data from Linux's *proc* [38] filesystem and can be extended by installing or development more PMDAs to support other resource information sources [39]. PCP centralizes all the data and identifies it with a canonical name to group similar metrics by type. A single metric may have more than one instance, for example, a network adapter or disk device, as multiple devices may be present on a Compute host. Each instance has its unique identifier, this ID may variate by host. Metrics that by definition are a single instance, such as total CPU utilization, use a default ID.

Optional PMDAs for Docker monitoring are available that specifically target Docker containers, but we opted to use the CGROUP based metrics from PCP as it enables a simpler PCP configuration in each compute node and the ability to support other container engines. The Docker extension for PCP didn't provide any new metrics that weren't already available in PCP.

PCP provides the ability to store a configurable set of metrics to a binary file with the use of the PMLOGGER service. This data can then be extracted and analyzed. For our container migration platform, we decided to not use this feature, as we need real time resource information and prefer to centralize the monitoring configuration on a single host.

Performance co-pilot’s REST API works by requesting the server with the name(s) of the metric(s) to monitor. The server answers back with a JSON data payload containing the name of the metric, a timestamp and all instances of the metric with their respective value. To access the server, a PCP Context token must be generated first. An initial request is sent to the server to create a new Context token and subsequent requests must be sent using this token as parameter. When monitoring containers another initial request must be sent to lock a PCP Context with a container name. The PCP REST server replies back with metrics only for that container’s CGROUP.

Performance Co-pilot can be deployed in different configurations enabling different types of network architectures to be tested. The main difference between deployment models are the PCP services running on the host to be monitored. The minimum services needed to provide distributed monitoring are the PMCD service running on the host to be monitored and a PMWED on the network. The PMWEBD can run on the same host to be monitored, this model is the per-node PMCD and PMWEBD (Figure 4-1)

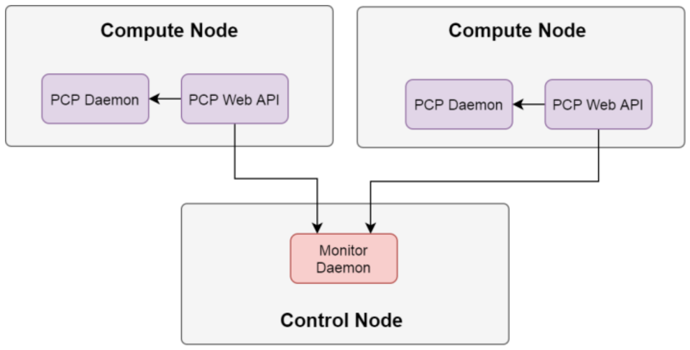

Figure 4-1 Per-host PMCD and PMWEB.

This model provides higher granularity of control for monitoring and simpler configuration, with the trade-off of the possibility of increased overhead on the compute host.

The system can be configured to use a proxy to connect directly to the PMCDs on the compute hosts and only have a single PMWEB on the network. This model is the per-host PMCD with central PMWEB (Figure 4-2).

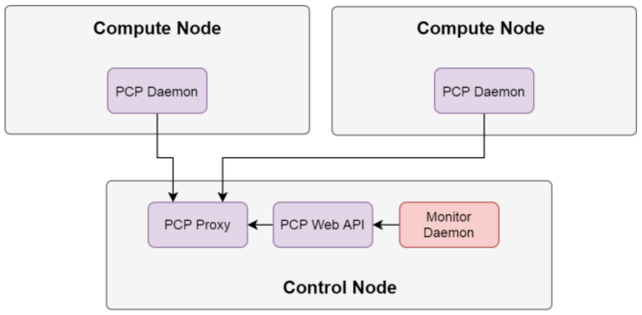

Figure 4-2 Per-host PMCD, central PMWEB.

This model provides a simpler compute-node software deployment with the possibility of lower overhead, but with a higher complexity of configuration and increased latency.

We tested both PCP deployment models by generating requests to the PMWEBD process. To test the Per-host PMCD, central PMWEB. we set the Control-host's PCP configuration as the main PCP REST server by modifying its local configuration. We also changed the PCP daemon configuration on the compute nodes to enable central PMWEBD polling. Our small-scale tests show, that no performance benefit was gained by using this deployment model, compared to the initial setup and default PCP configuration of Per-host PMCD and PMWEB PCP deployment model. Using the Central PMWEBD model did show some degree of overhead but minimal. The cost and complexity of setting up the central PMWEB model was not worth it for our setup and workloads. We keep our original Per-host PMCD and PMWEB configuration and tweaked the PCP server timeout values to better suit our workloads and resource sampling rate.

4.2 HerdMonitor OVERVIEW

4.2.1 Monitoring Daemon

In our proposed live migration platform, the Control node runs the system monitoring daemon that polls the PMWEBD server to gather resource utilization metrics from the compute-nodes and containers. It keeps track of each container on the system and its current host during container live migration and is able to actively collect performance metrics. We developed a system monitoring daemon called Herd Monitor that connects to PCP to gather metrics, keep track of container migrations, and deliver real-time data to the Control Daemon.

4.2.2 Configurable Options

Herd Monitor can be configured to monitor any metric provided by Performance Co-Pilot (PCP). A host-list configuration file defines all the compute hosts to monitor based on their fully qualified domain name or their IP address. A single configuration file contains the list of performance metrics names to monitor for all compute host. On this file, each metric name also contains a list of operations to be applied to the metric, as well as the data type of the metric. An additional configuration file per individual host is set that contains a list of performance instance domains per metric name to monitor. If a compute host contains more than one device, for example disk drive or network adapter, the specific device to monitor is set on this file as an instance domain. Herd Monitor can also be configured to monitor other hosts that are not compute hosts. These hosts do not track containers or their migrations.

For all containers, a single configuration file is set. This file contains the metric names to monitor as well as the data processing operations to apply to each metric and its datatype. Herd Monitor allows the configuration of independent performance metric sampling rates for containers, hosts and meta-hosts as well as the rate in which the container actions, such as container added or removed, must be tracked as command line parameters.

4.2.3 Metric Processing

Herd Monitor does real time processing of the monitored performance metrics given the configuration file that describes the metric and operations to apply. These operations are customizable and range from metric conversion of bytes to kilobytes, time conversions such as nano seconds to seconds. The metric processing also takes care of applying cumulative transformations to change nominal values to interval values of cumulative counters by doing a derivate over time.

4.2.4 Data Output

All information is logged in comma separated value (CSV) formatted files for later analysis. A CSV file is generated with the gathered metrics for all compute hosts. A separate file is generated with the metrics for all containers regardless of compute hosts and grouped by a universal identification for each workload. Another CSV file is generated with the actions of each container on each host, such as added or removed to each host; used to keep track of migrations. For each container, a single log file is generated that contains the standard output of each workload running on said container, this file can be parsed and used to validate integrity of the system.

4.2.5 Optional Metrics

Aside from the performance metrics provided by PCP, each generated sample contains general information such as Compute host fully qualified domain name (FQDN), container name, time in Unix time format. Herd Monitor can merge additional metrics provided by a running daemon on the Compute node. For our system, we stream additional container information not available in PCP from the Compute node to the monitoring daemon using MQTT. We added this functionality to our Compute daemon but a simple MQTT client can stream additional information.

4.2.6 Herd Monitor Architecture

Herd Monitor is a multi-threaded software written in Java, its purpose is to gather real-time resource data, mostly from PCP (Figure 3-2); its main components are:

- Daemon thread: is the main thread of the program that starts all other threads
- Metrics Processor thread: processes raw metrics, it computes transforms and unit conversions.
- Host Monitor thread: monitors the Compute nodes' hosts performance metrics using the PMWED
- Migration Tracker thread: keeps track of container actions, such as added or removed to keep track of container migration throughout hosts. This thread lets the Container Monitor thread know what container to monitor and which host to poll the data from. It keeps track of migration count per container and runs at a faster sampling rate than the Container Monitor thread to handle very quick container migrations.
- Container Monitor thread: monitors the containers' performance metrics of a container running on a given Compute node using the PMWED. It needs to keep track of the current control group (CGROUP) assigned to the container per host to monitor the containers.
- Container action queue: generated by the *migration tracker* thread and used by *container monitor* thread to keep track of what containers to monitor on each host. It enables Herd Monitor to support container live migration.
- Remote Logger thread: receives the standard output of a container from their current compute host through MQTT and saves the output to a single file regardless of current Compute host.
- Message Queue thread: keeps a connection to the MQTT broker to stream real-time data to other MQTT clients. This thread also handles optional metrics sent by the Compute node.

- File loggers: outputs the monitored resource data to an independent comma separated files (CSV) by Compute host, container, migrations, operations and container standard output. These files are named by the timestamp Herd Monitor was started.

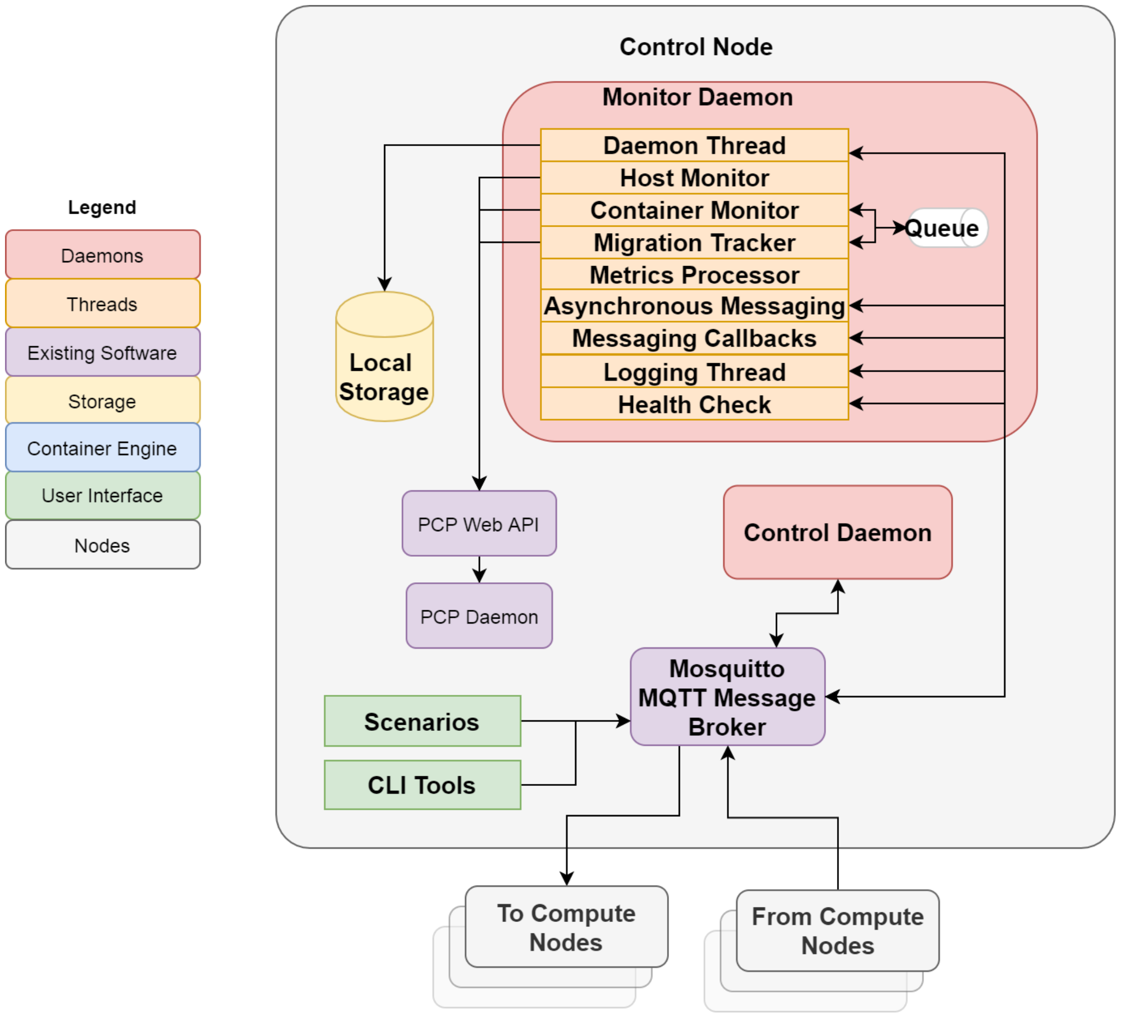

Figure 4-3 Detailed Herd Monitor with its threads

See **Appendix I – Herd Monitor Pseudocode** for the pseudocode of the threads found in Herd Monitor

4.3 SYSTEM CHARACTERIZATION & DISCUSSION

4.3.1 Developing Herd Monitor

The monitoring daemon development processes started by initially monitoring Compute nodes resource information, later expanding to containers and container migration tracking between Compute hosts. With the initial host monitoring test, we compared results with other existing monitoring tools such as *htop* for CPU and memory utilization, *iotop* for disk metrics and *speedometer* for network metrics to validate the generated output. We developed the metric configuration files for each compute host using the datatype and unit type information (such as instant metric or counter metric) from PCP, using the tool *pminfo*. To aid in the creation of those container and host's configuration files, we developed a script to gather all the instances of the metrics that could have more than one instance domain. For container monitoring, Herd Monitor uses a base configuration file and does not require a definition of instance metrics, as the request to the PCP REST server is unique (by PCP context token) for each container.

Each metric that is defined by PCP as a *counter* metric is an incremental value that needs to me *cumulatively transformed* from the previous sample to get the change. We do not include the output of the first received metric so we can compare to the second received metric and define a baseline. In the rare cases where a *cumulatively transformed* metrics results in a negative value, the whole metric sample is discarded as to avoid post processing data cleanup and erroneous samples to the Control node.

See **Appendix II – Herd Monitor Technical Challenges** for a writeup of the technical difficulties we found during the development of Herd Monitor

4.3.2 Overhead characterization

The monitoring sampling rates have a direct effect on the overhead of Compute nodes. The use of PCP to gather resource utilization data from the containers and the Compute nodes proved to affect the CPU utilization of the Compute nodes up to 10% in some cases. The overhead on the Compute nodes affects the performance of the containers and their workloads. PCP requires an access token to remotely monitor hosts called PCP context. To monitor containers, a new context needs to be created for each container. The combination of new context creation on each migration, a high sampling rate for containers and Compute nodes, increased the overall system overhead affecting system CPU utilization and disk usage (Figure 4-4).

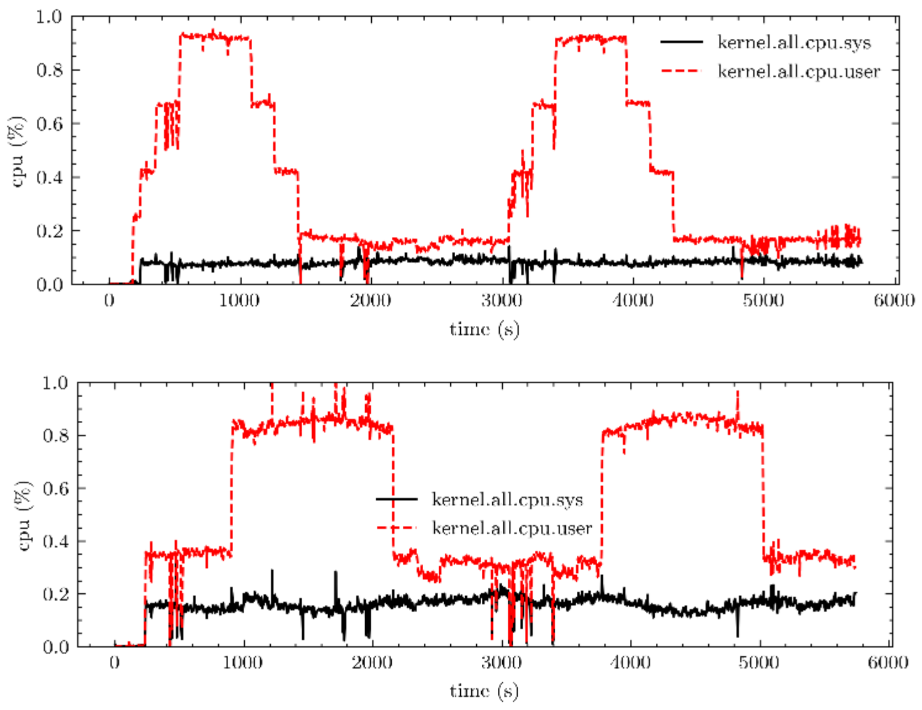

Figure 4-4 Two compute-node (top and bottom) CPU utilization for a staircase scenario testing CPU intensive workload showing signs of heavy system overhead caused by the monitoring software. Host CPU utilization normalized from 0-1 based on the number of CPU cores on the Compute Hosts.

To address this overhead, we studied the PCP deployment models and REST API parameters to explore how we could optimize our monitoring software to reduce overhead while keeping functionality.

4.3.3 System Optimization

We started optimizing by removing any unnecessary services running on the Compute nodes as well as the Control nodes. For the purpose of real-time monitoring PCP services, like the binary logger (PMLOGGER), are not needed. Another PCP service not required is the inference engine for performance metrics PMIE, as herd monitor is already performing some of the actions it provided in a minimal matter and running outside the compute-node. Any Linux services not required were also shutdown.

We tested different sampling rates to monitor the Compute nodes and containers, settling with sampling the containers and hosts at a rate that proved to be enough for workload analysis while reducing overhead. The monitoring software was optimized by reusing PCP connections and contexts, discarding processed data of migrated containers and modifying some events to run asynchronously. We split the tasks into containers monitoring, Compute node monitoring and the migration tracking to their own threads. Data concurrency management, data synchronization and resource locking were heavily tested to avoid race conditions and sampling aliasing.

To test these improvements, we added to Herd Monitor the ability to monitor the hypervisor host, as our testbed is virtualized, that runs the Compute nodes in our experimental setup, the meta-host. By monitoring the meta-host, we were able to discard the possibility of resource starvation from the meta-host as a reason to the system overhead present on the

Compute nodes that reduced the performance of the containers' workload. With these improvements we were able to reduce system overhead significantly (Figure 4-5)

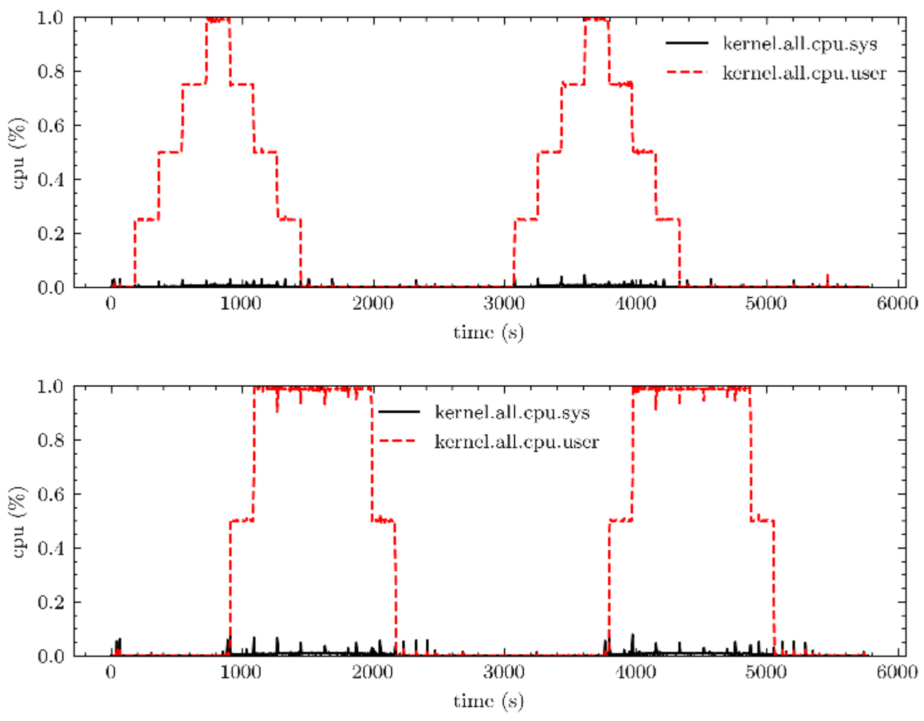

Figure 4-5 Two compute-node (top and bottom) CPU utilization on a staircase scenario testing CPU intensive workload after the system overhead mitigations and changes. CPU utilization normalized from 0-1 based on the number of CPU cores on the Compute Hosts.

4.3.3.1 Monitor Daemon error handling

Herd Monitor can handle network connectivity errors, PCP context error, and will retry the connections to the Compute nodes. The monitoring daemon also handles resource metric processing errors and migration tracking errors. The Control daemon in the Control host handles MQTT errors, missing data and the possible uncorrectable migration errors from the compute daemon.

Herd Compute's Remote logger thread also takes care of handling *Status* Messages. Each module on the Herd system that uses MQTT sends a status message to announce itself. In the case of disconnection, each module is set to send a *Last Will and Testament* [42] message that is automatically published. In the case of module reconnection, every time a client subscribes to a topic, a connection message is sent to the monitoring daemon that can replay messages back. Once of those cases is used to log a container, as a *LogStart* message is sent from the monitoring daemon to the compute daemon. If the compute daemon reconnects, the monitor daemon will send again that message to the compute daemon, to restart the Logging thread.

4.3.4 Testing

4.3.4.1 Host Monitoring

Host monitoring was tested by running containers within each host and observing its behaviors on each host (Figure 4-6). We set Herd Monitor *hostlist* to monitor over the network multiple Compute hosts.

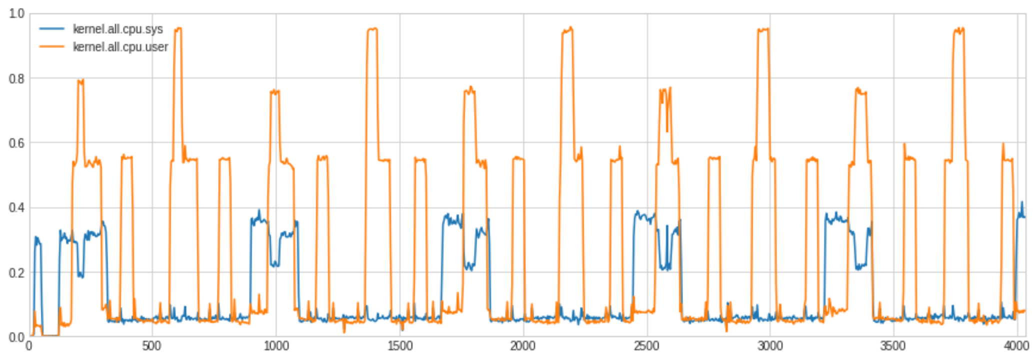

Figure 4-6 Initial results for Host monitoring, CPU system and user utilization over time. Host CPU utilization normalized from 0-1 based on the number of CPU cores on the Compute Host.

4.3.4.2 Container Monitoring

Synthetic workloads such as Sysbench and Stress-ng are run within containers to test all the complete functionality of the system. These workloads are highly customizable and can be used to emulate different types of workloads such as online transactions processing (OLTP). They were used to have an initial understanding of the systems performance and later were used validate the resource metrics gathered by Herd Monitor. Some of the initial tests was running a memory Sysbench test on a container and migrating it to four different compute hosts sequentially (Figure 4-7).

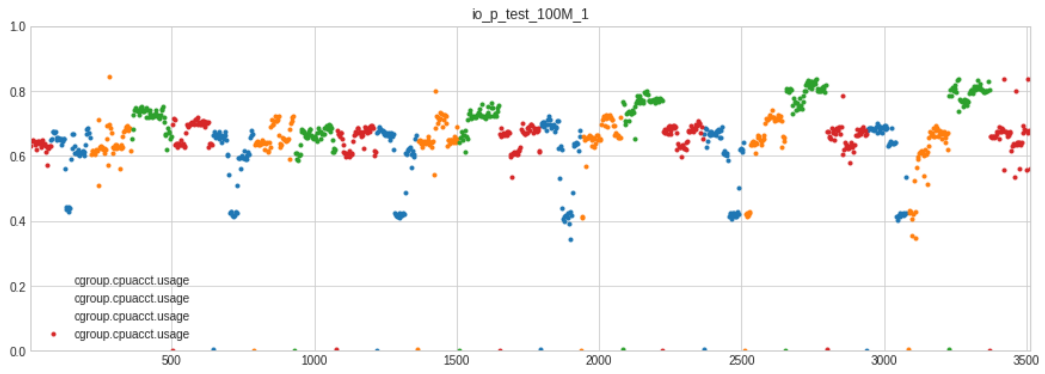

Figure 4-7 Initial results for live-migrating Container monitoring. Each color represents a different host for the same container. Container CPU utilization over time normalized from 0-*N* based on the number of *N* CPU cores on each Compute Host.

4.3.4.3 Migrating Container Tracking

To test container migration tracking a live migration stress test was performed. This test was used to understand system resilience, performance, overhead and resource capacity. These tests were performed using the sequential migration tool and keeping the container as little time as possible in a compute-node without causing sampling aliasing on the monitoring tools. We tested different tracking rates to monitor the container actions and settled with tracking rate of 2 seconds. This rate proved to be high enough to handle quick migrations while not inducing system overhead. For migration tracking, we plot a blue line that represents the moment in time a container was added, and orange line represents a removed container from a compute host (Figure 4-8).

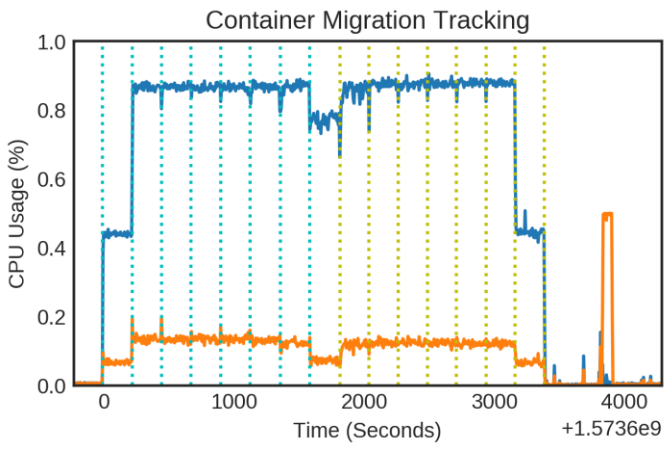

Figure 4-8 Live-migrating container tracking. Host CPU utilization over time normalized from 0-1 based on the number of CPU cores on the Compute Host.

4.3.4.4 MQTT Data Streaming

Real time data streaming was originally tested on a different Herd development branch and then merged to the main code. MQTT streaming was developed by using the same framework as the one on Herd Compute and tested using Mosquitto client tools[43]. We tested different quality of service (QoS) priorities and settled with at-least-once for monitoring streaming and exactly-once for status and control messages.

4.3.4.5 Container Standard output logging

Herd Monitor uses the MQTT features to subscribe to each Compute-node MQTT broker to get standard output (stdout) published by Herd Compute. The Remote Logging thread handles Herd Compute reconnections, merges container logs for the same container as it migrates to other hosts and avoids data duplication by using a nano-second timescale timestamp (generated by Docker) in the case or reconnections as the Compute Daemons sends the entire log to the monitoring daemon.

4.3.4.6 Monitoring Testing Methodology

Workload monitoring, container tracking and system overhead was tested on the system using controlled, repeatable experiments we call scenarios. These scenarios are a set of scripts that automate sequential container migration. A base scenario consists of starting all daemons and services, letting the compute-nodes reach a baseline and remotely launching containers running a synthetic workload. Once the containers are launched and its baseline reached the script starts migrating containers sequentially to other compute-nodes. Every property of a scenario for instance, time between migrations, can be customized. Different scenarios are used to test different aspects of the system such as sampling rate for monitoring, sampling window for workload analysis, migration stress testing, resource starvation among other possibilities. Each scenario may require a new hardware configuration, in those cases, the compute-nodes virtual hardware configuration is changed accordingly. New hosts can be used for resource-heavier scenarios such as memory intensive tests or testing RAID disk configurations.

4.3.5 Sampling rate comparison

To test the effects on data resoultion of different sampling rates, we ran the same scenario multiple times with different sampling rates to observe the granularity of the results. (Figure 4-9) We settled for 5 second sampling as it provided with enough samples for short workloads to conduct workload characterization analysis. This test was conducted before the overhead optimizations were applied to Herd Monitor.

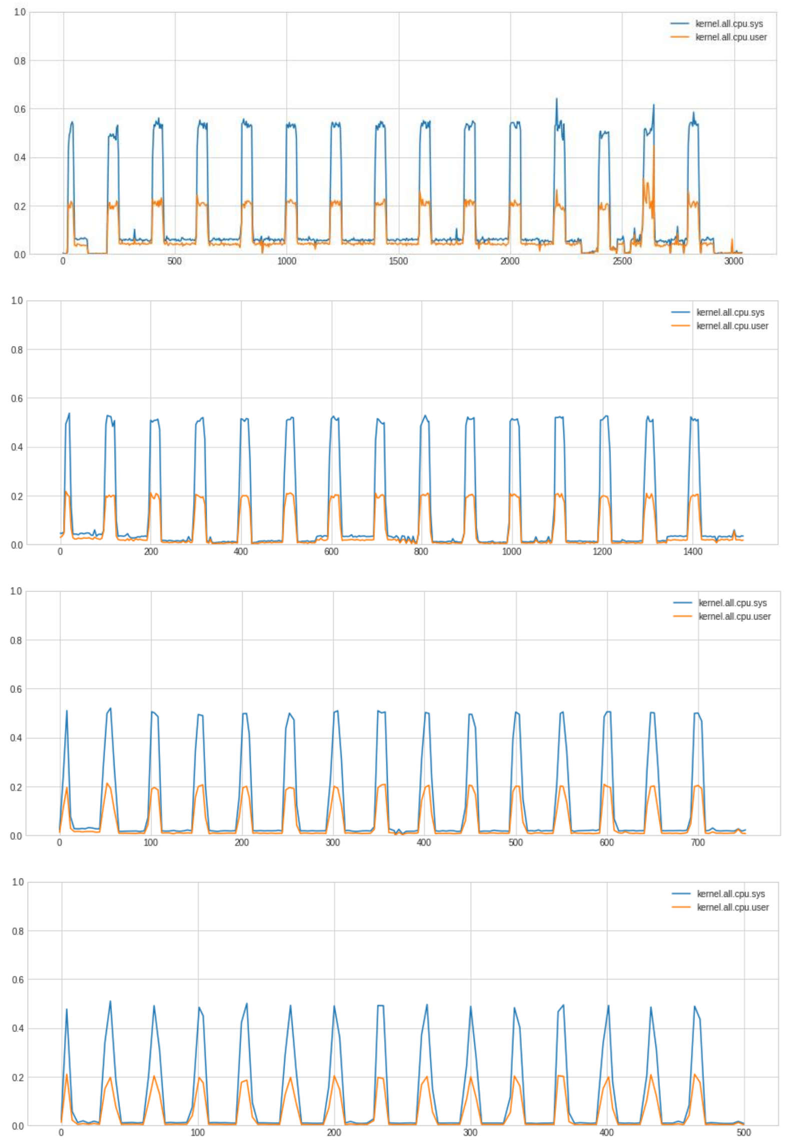

Figure 4-9 Host CPU utilization over time normalized from 0-1 based on the number of CPU cores on the Compute Host on a workload monitored with 5,10,20,30 second sampling rates

4.3.6 Overhead and Scaling

As one of the main goals behind the development of Herd Monitor is for it to be lightweight, we tested its overhead on the system by isolating Herd Monitor on a Control host and running it for one hour while monitoring the Control host. For our workloads, we settled with 5 second sampling rate, and for migrations we settled for 2 second tracking rate. Our tests showed that this combination provided with enough samples while keeping a high metric granularity and low system overhead. (Figure 4-10)

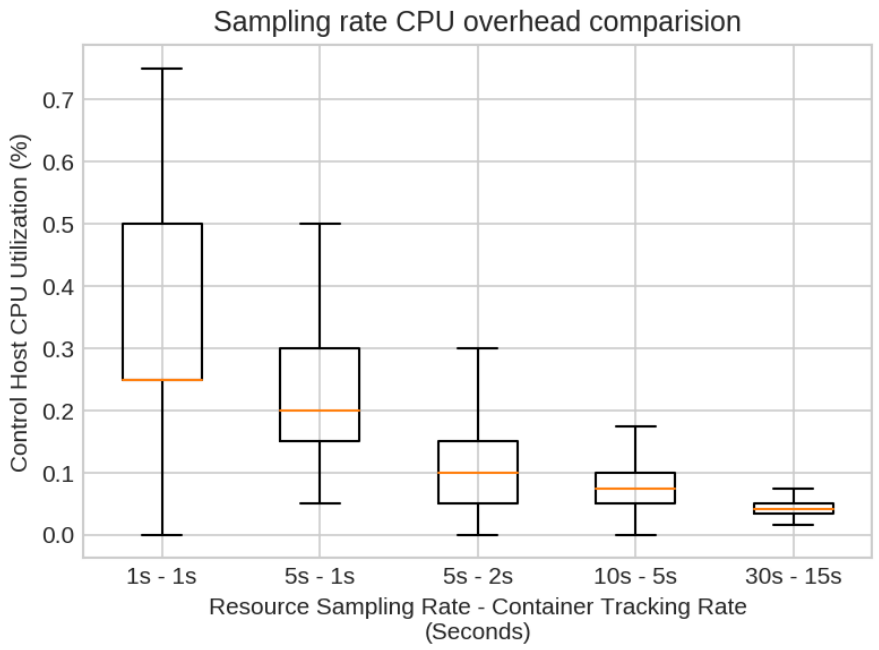

Figure 4-10 Sampling rate CPU on Control host overhead for diffident sampling rate configurations. Host CPU utilization normalized from 0-100 based on the number of CPU cores on the Control Host.

We tested scaling of Herd Monitor by deploying hundreds of containers over six compute hosts (Figure 4-11). Our initial tests show that the Java virtual machine (JVM) was the main limiting factor. After increasing the available memory to the JVM, the system memory became a bottle neck. To test scaling, we deployed 25 containers at a time until the compute host or the control host reached its limits (Table 4-1). We ran each test without a load (empty containers) and under CPU load (Sysbench 1 thread CPU test). After reaching an amount of 2700 containers monitored at the same time, PCP started having errors, and dropping samples for the next amount of containers until reaching out test limit of 3600 containers on the compute node. The PCP errors where related to the context timeout for each request to PCP as we tested scaling using 5 second sampling rate and PCP started showing latency issues and timing out, sending error responses to Herd Monitor. After changing the timeout out from 10 seconds to 60 seconds, latency improved, and Herd Monitor reached a total of 2884 containers under monitoring.

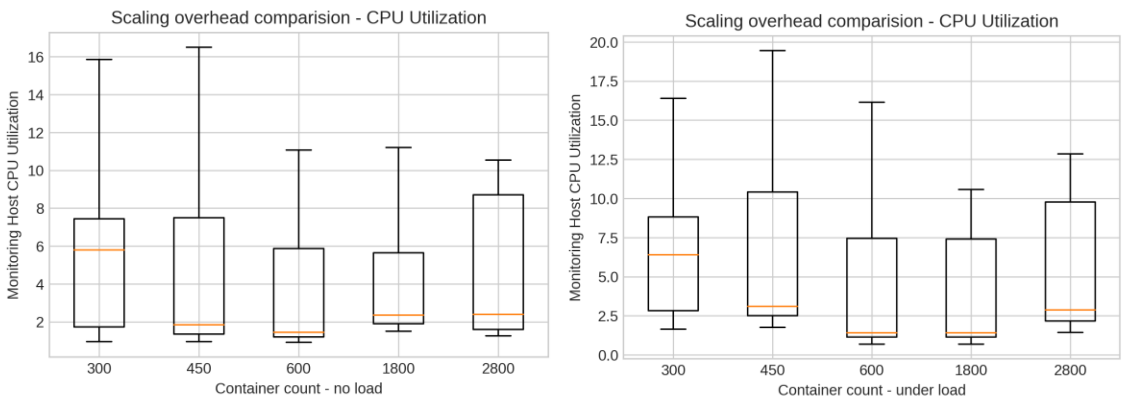

Figure 4-11 CPU overhead on the compute node for different amount of containers monitored and hardware configurations. Host CPU utilization normalized from 0-100 based on the number of CPU cores on the Control Host.

Monitoring Host System Memory	Herd Monitor JVM Memory	Total Simultaneous Monitored Container Count
4GB	1GB	300
4GB	4GB	450
16GB	8GB	600
16GB	16GB	1800
16GB	16GB	2800

Table 4-1 System configuration for scaling tests

4.3.7 Distributed Monitoring

Herd Monitor can be configured to work in a distributed manner by setting up multiple monitoring daemons in separate host to communicate with a central control daemon. This setup should allow the monitoring of higher container counts. Our tests show that monitoring 75 containers per compute hosts is stable (Figure 4-12). As the main bottle neck is PCP (Figure 4-13), this can be mitigated by lowering the sampling rate in the case of many containers.

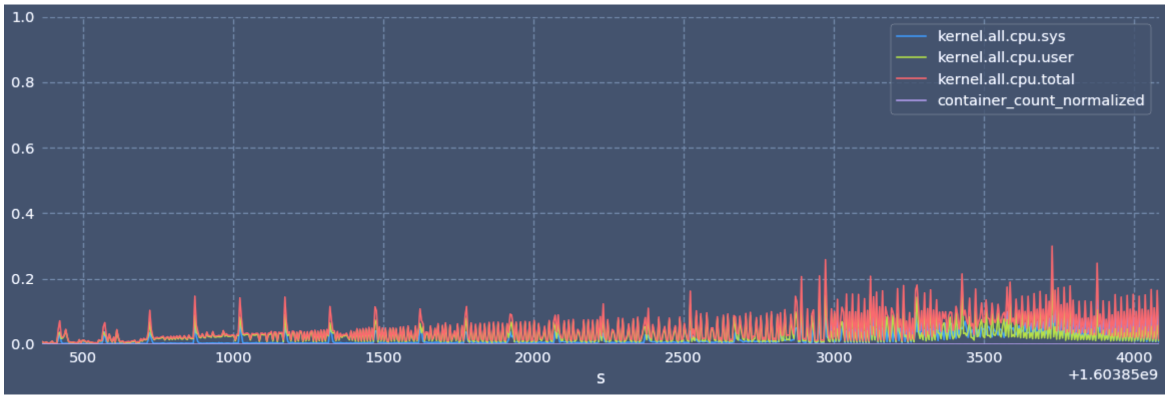

Figure 4-12 Host CPU utilization normalized from 0-1 based on the number of CPU cores on the Control Host monitoring from 0 to 3600 containers

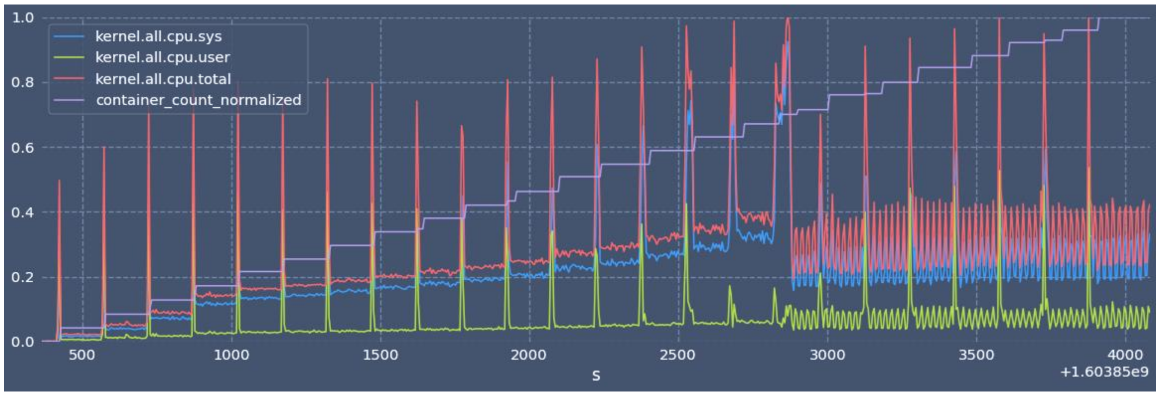

Figure 4-13 Host CPU utilization over time normalized from 0-1 based on the number of CPU cores on a Compute Host reaching system limits

5 Resource Provisioning

To provide resource provisioning to containers by the use of live migration, Herd implements container placement policies. We identified important resource metrics for each workload that aided us in the development of these policies that brings better performance to the system through container live migration. We incrementally tested different workloads and implemented multiple migration policies. We used machine learning for feature extraction by analyzing the resource utilization data of workloads coupled with validation results to get to an ideal resource provisioning policy for an Online Transactional Processing workload (OLTP). We also used machine learning for performance forecasting of a container's workload.

5.1 Workload characterization

To have an initial understanding of the system's performance under load conditions, we ran a series of experiments to evaluate the characterization of different workload types. The experiments helps us understand the performance impact of container live migration for CPU intensive, Memory intensive and Disk (IO) intensive workloads. Workload characterization helps us identify correlations between the different resource metrics, monitored with Herd Compute, for each workload type.

Workload monitoring, container tracking and system overhead was tested on Herd using controlled, repeatable experiments we call scenarios. These scenarios are a set of scripts that automate sequential container migration. A base scenario consists of starting all daemons and services, letting the compute nodes reach a baseline and remotely launching containers running a synthetic workload. Once the containers are launched and its baseline reached, the script starts migrating containers sequentially to other Compute nodes. Every property of a scenario for instance, time between migrations, can be customized. Different scenarios are used to test different aspects of the system such as sampling rate for monitoring, sampling window for workload analysis, migration stress testing, resource starvation among other possibilities.

For workload characterization we developed a series of scenarios where container execution overlaps in time with other containers, we call these overlap sequences *staircase scenarios*. They are used to understand Compute host resource depletion and how the workload behaves on those cases. The containers run synthetic workloads such as *Sysbench* and *Stress-ng*. These workloads are highly customizable and can be used emulate different types of workloads such as OLTP.

We deployed a testbed that includes all the developed Herd modules and open source software, installed on virtual machines running Ubuntu 18.04 over a VMware hypervisor. We named the baremetal machine that runs the hypervisor, which hosts compute nodes, the *meta-host*. A multi meta-host setup was tested by extending the VMware Virtual Network with the addition of a *pfSense* virtual machine that acts as a router. Using different meta-hosts allows the system to be tested under different hardware configurations. Each scenario may require a new hardware configuration, in those cases, the Compute nodes virtual hardware configuration is changed accordingly. The use of a visualized testbed enables the system to be modified to add and remove more Compute nodes running on different hosts. More Compute nodes can be added to the system by cloning a compute node VM and changing the Herd Monitor configuration. New hosts can be used for resource-heavier scenarios such as memory intensive tests or higher CPU core count configurations. Four compute node VMs were used to run the initial workload characterization experiments.

5.1.1 CPU characterization

Sysbench was used for the CPU characterization experiments. It provides us with the ability to change thread count and execution time. Sysbench's benchmarking capabilities for Linux supports testing of CPU, memory, file I/O, mutex performance, as well as OLTP using MySQL as server.

We started testing container overlap by setting 4 containers over a 4 core CPU compute host to evaluate CPU utilization scaling. We then migrated the containers to a 2 core CPU compute host to understand how the resource metrics behave under a new host. With this test we can observe how container CPU utilization affects the compute host's CPU utilization. (Figure 5-1) We also identified positive correlations between the metrics monitored such as *kernel load* with the number of processes in the *runnable* state.

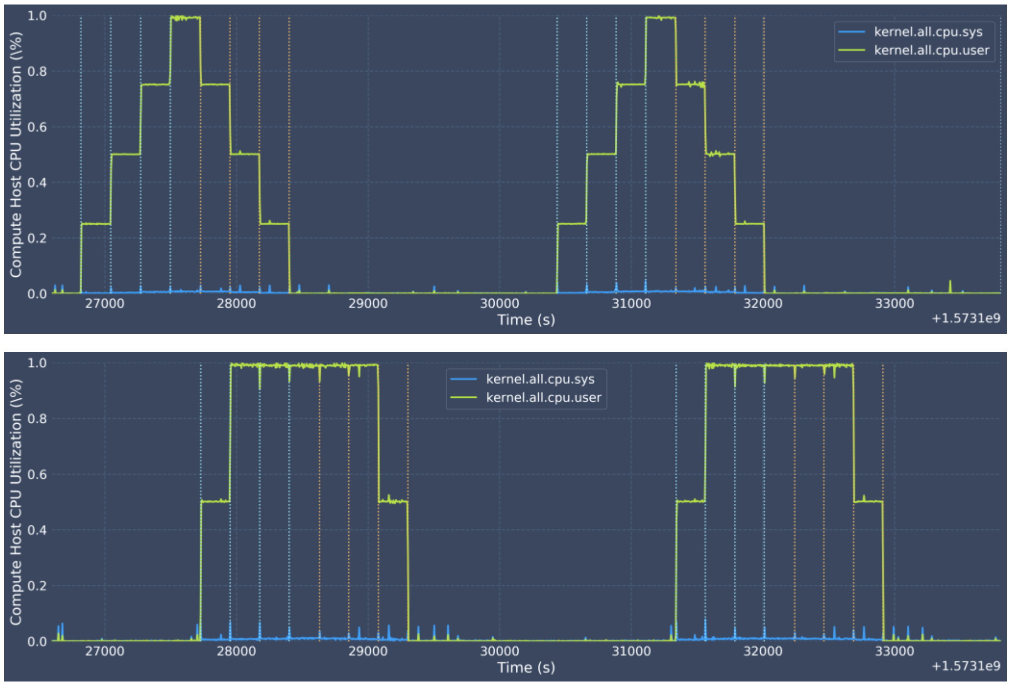

Figure 5-1 Host CPU Utilization over time normalized from 0-1 based on the number of CPU cores on the Compute Host on a 4 core and 2 core Compute Hosts

With the CPU workload characterization experiments, we studied the effect of single threaded and multi-threaded workload as it overlaps with other containers on the same host. We evaluated the effects of migrating containers to and from hosts that don't have enough resources to support the workloads. (Figure 5-2) We also understood the CPU-time scheduling of multi-threaded workloads and single-threaded workloads on the same resource-depleted host

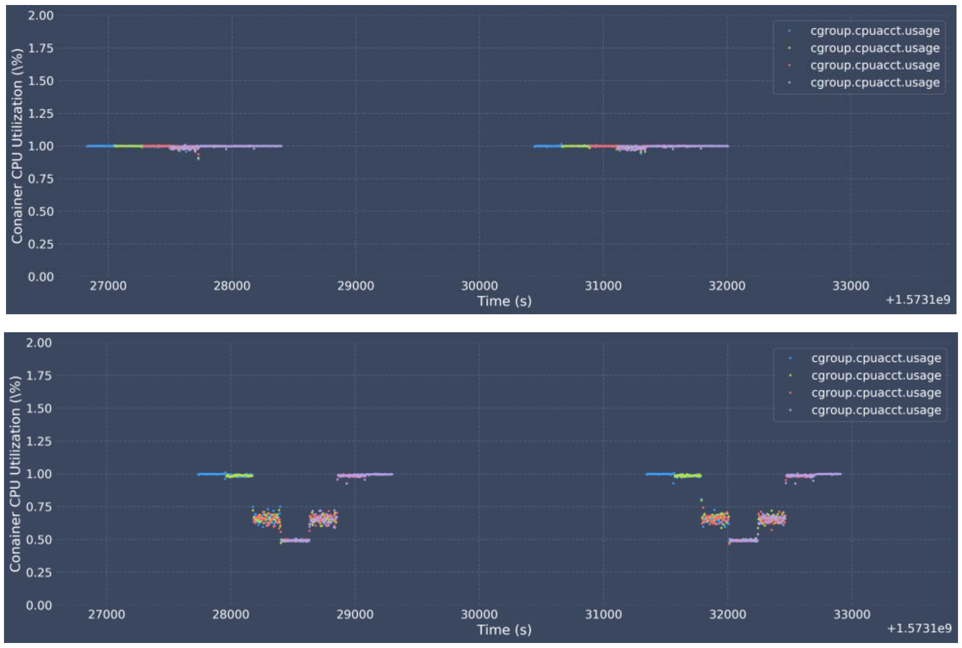

Figure 5-2 Container CPU utilization over time normalized from 0-1 based on the number of CPU cores on each Compute Host. 4 core and 2 core Compute host

CPU utilization for containers is measured as non-normalized host CPU utilization, that is without taking into account the Compute host CPU core count. A double-threaded workload's baseline is 200% while a single-threaded workload is 100%. This utilization can then be normalized to host CPU utilization by dividing it by the host CPU core count. The host's scheduling algorithm (CFS) distributes CPU utilization evenly to the containers even if they have a higher thread-count. CPU utilization normalization can be enabled or disabled for hosts and containers in the Herd Monitor metrics configuration file. We decided to keep the Host CPU utilization normalized by host CPU core count while keeping the containers CPU utilization not normalized. This allows us to better understand how a containers performance scales over different compute hosts. With the CPU tests we can visualize the performance drops on each container based on its available CPU time and the resource utilization gains when moved to a host with available resources. (Figure 5-3)

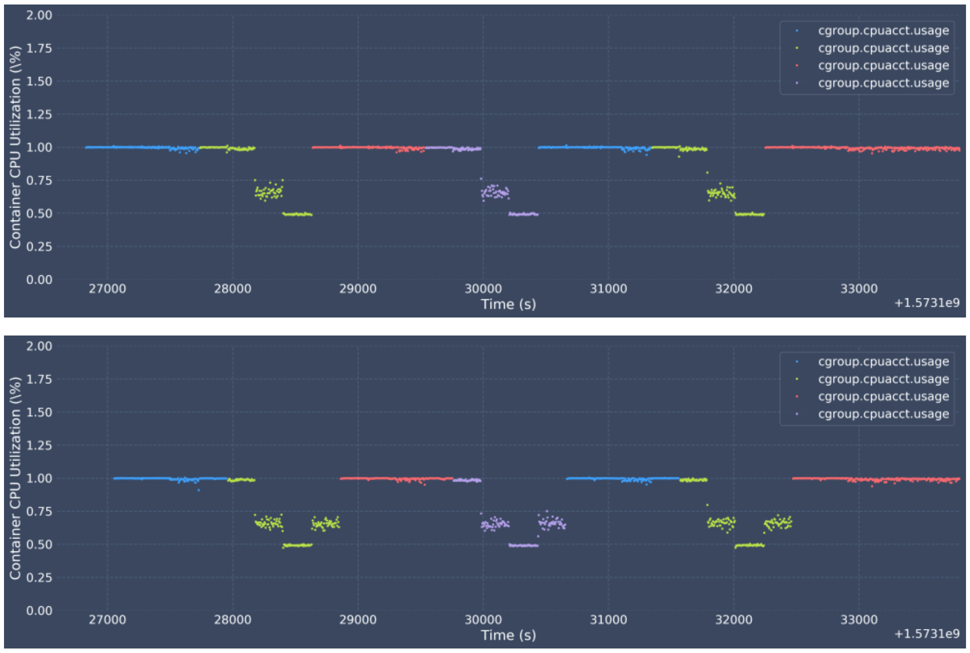

Figure 5-3 Container CPU utilization over time normalized from 0-N based on the number of N CPU cores on each Compute Host. Two containers over four different compute hosts

5.1.2 Memory Workload characterization

For Memory-based workload characterization we follow the same steps we did for CPU-based workload characterization using the stress test software called Stress-ng. It tests various computer's physical subsystems as well as operating system kernel interfaces. We ran a *staircase* scenario with 4 virtual memory workloads of 256MB of size on containers hosted on a compute node with 1GB of RAM. The purpose of this experiment was to force the system into memory resource exhaustion so we can observe system memory paging behavior during live migration.

As the compute nodes have a small amount of system memory, the host reaches memory saturation by the time the 3rd container is launched (Figure 5-4). The host enters severe paging, visible by the high major faults per second the system required loading a memory page from disk and high Kernel CPU utilization. When a container is migrated to another host the pagefault count decreases and User CPU utilization increases (Figure 5-5). We identified positive correlations between Kernel CPU utilization and pagefault count and software Interrupts (Figure 5-6).

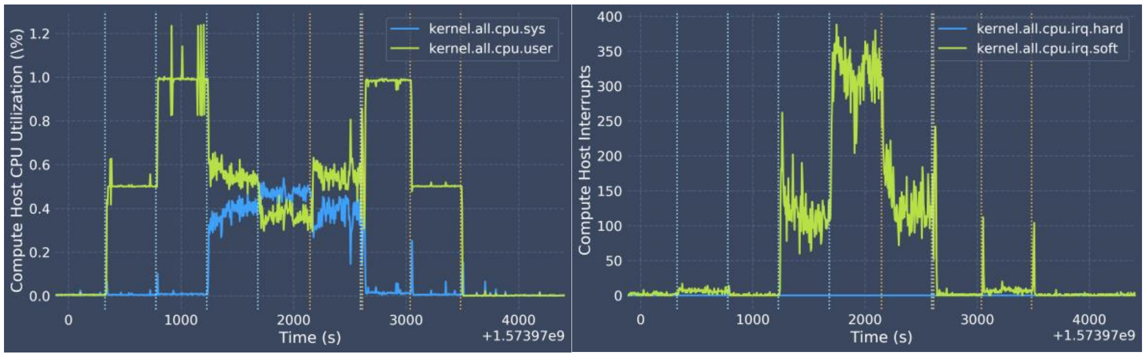

Figure 5-4 Host's CPU Utilization (left) Host Interrupts (right)

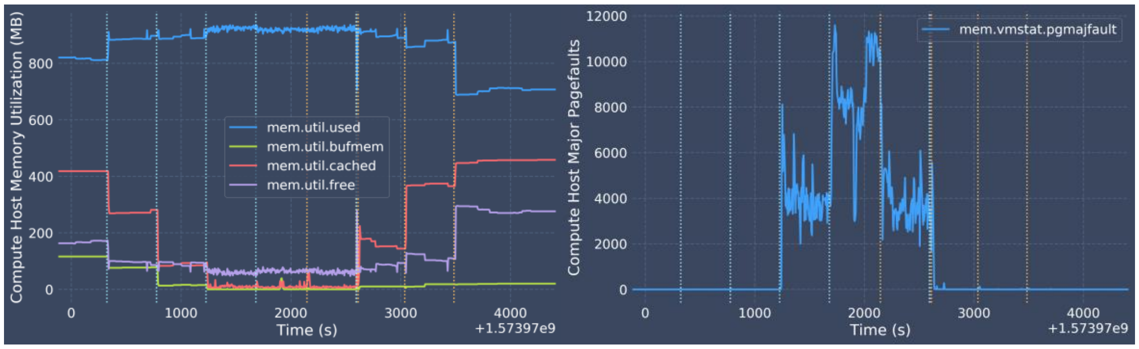

Figure 5-5 Host's Memory Utilization (left) Host Pagefaults (right)

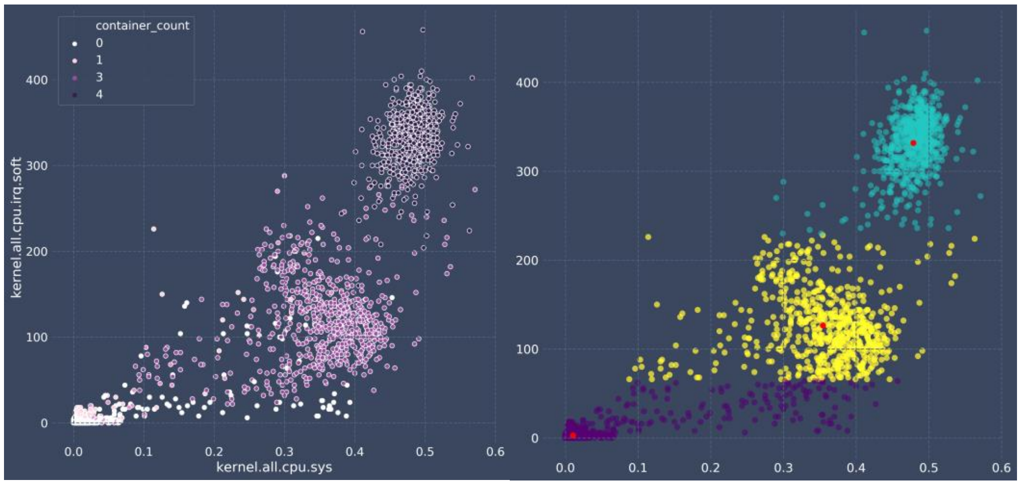

Figure 5-6 Hosts Interrupts vs Kernel CPU utilization and Container Count as color (Left) 3 cluster Kmeans over same data (right)

5.1.3 Disk IO Workload characterization

We ran a series of *staircase* scenarios using Sysbench IO tests with different configuration and sizes. Testing sequential and random read and write with different ratios with payloads of 100MB and 1GB. We found that that disk operations and disk throughput are closely related as well as system interrupts and CPU kernel utilization (Figure 5-7).

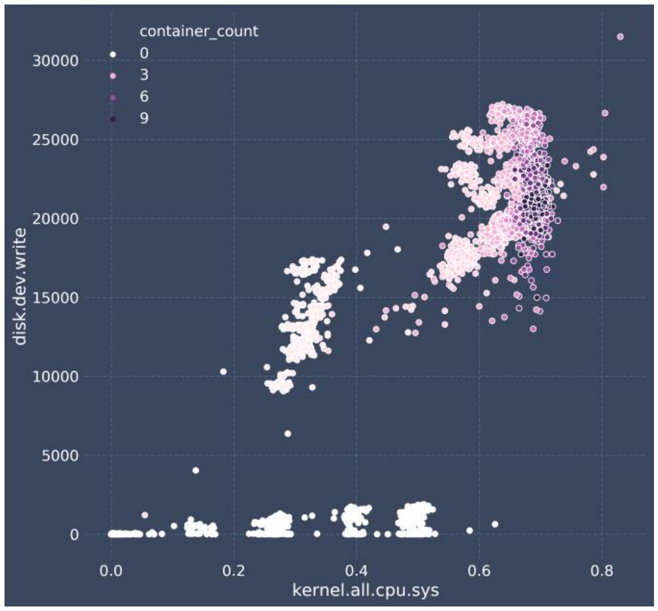

Figure 5-7 Host Disk Requests vs Kernel CPU utilization

5.2 Herd Control Overview

5.2.1 Control Daemon

The Control node also runs the Control daemon. Its main task is to use the data that was gathered and processed by the Herd Monitor from the Compute nodes and containers to make container placement and migration decisions. We developed a system resource provisioning module called Herd Control that processes the real time data a given timeframe and move the containers to another Compute host based on the active migration policy to achieve better overall system resource utilization.

Herd Control is a multi-threaded software written in Java, its purpose is to process real-time resource data, and send migration requests (Figure 5-8); its main components are:

- Daemon thread: is the main thread of the program that starts all other threads
- Message queue thread: keeps a connection to the MQTT broker to receive real-time data to the Monitor Daemon.
- Migration Controller: stores the real-time data in DataFrames to process it and send migration requests.

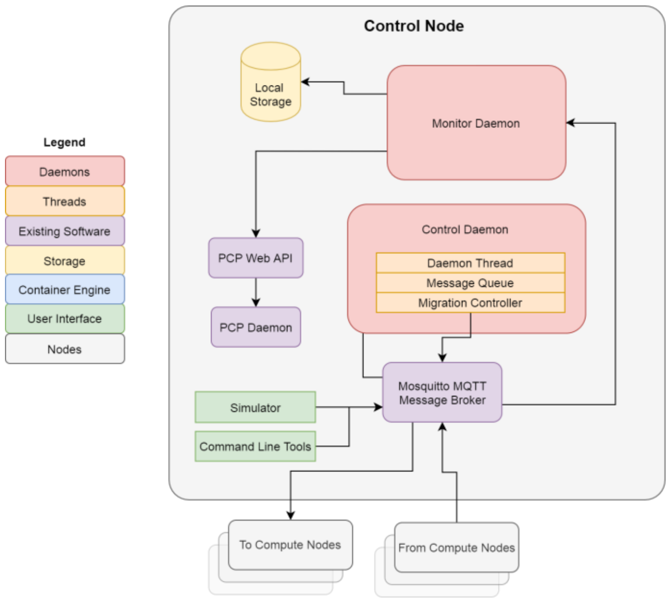

Figure 5-8 Detailed Herd Control with its threads

5.2.1.1 System Messaging

For container live migration, the Herd Control sends a message to a Compute daemon by publishing on its respective compute host's MQTT broker. The compute daemon on that hosts will start the migration processes by communicating with the target host (Figure 5-9) that will receive the migrated container to finish the migration process. During monitoring, the compute nodes will stream resource metrics to the control-node using MQTT.

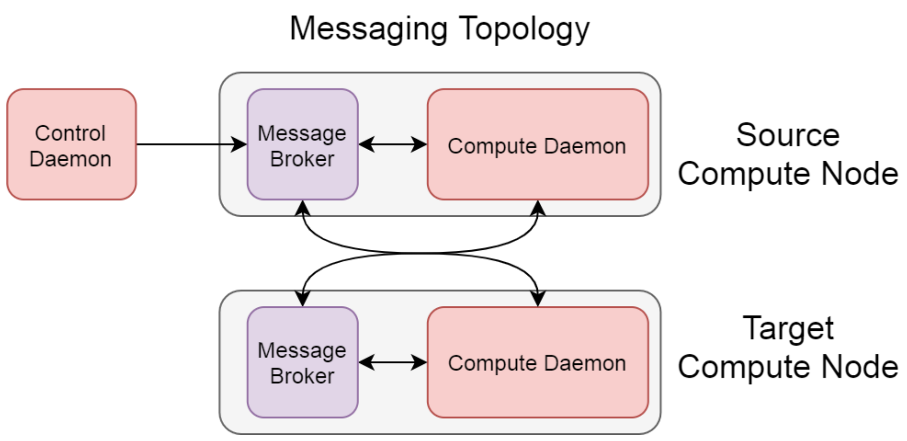

Figure 5-9 Herd Control Messaging topology

The system messaging can be summarized in:

- Container Migration: node to node
- System Monitoring: nodes to central
- System Controlling: central to nodes

5.2.1.2 Placement policies

Herd Control implements the container live migration policies that manage system resource provisioning. The resource utilization data streamed in real-time from Herd Monitor is used by the policies to determine container live migration placement thought the system.

5.2.1.3 Integrating all modules

To test system integration of all modules we developed a preliminary resource provisioning policy that live-migrates containers based on the data provided by Herd Monitor (Figure 5-10).

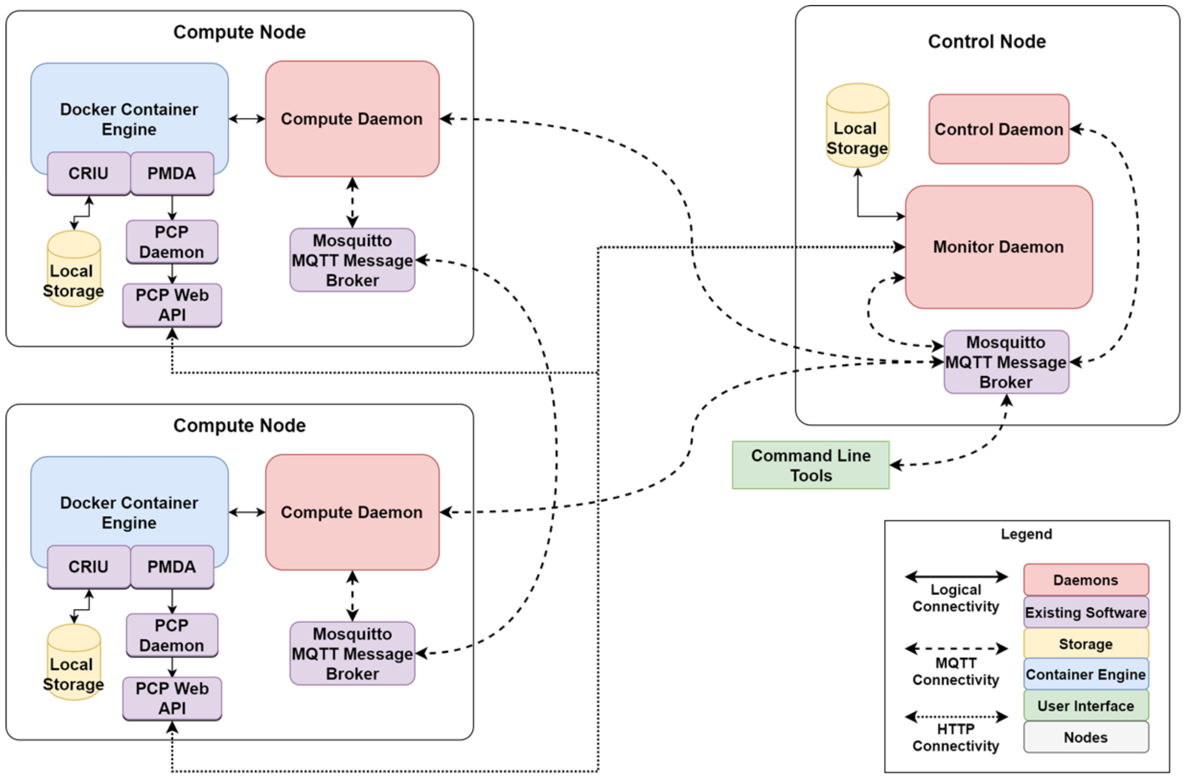

Figure 5-10 Herd system setup for two compute hosts with live migration support for resource provisioning.

5.3 Designing Container placement policies

5.3.1 Instant performance metrics

With the workload characterization experiments we understood how each workload managed resources. We also observed the possible benefits of container live migration to other hosts that have more available resources. The Sysbech workload provides the ability to report the instant performance metrics during execution, we used this to test our hypothesis that more resource utilization translates to better performance. To get this data in real time from each container we modified Herd Control to parse and filter in real time the containers standard output and stream over MQTT to Herd Monitor as another resource metric. We call these metrics *ops*, as in operations per second. We then evaluate each workload with its respective workload performance report to validate the actual performance of each test (Figure 5-11).

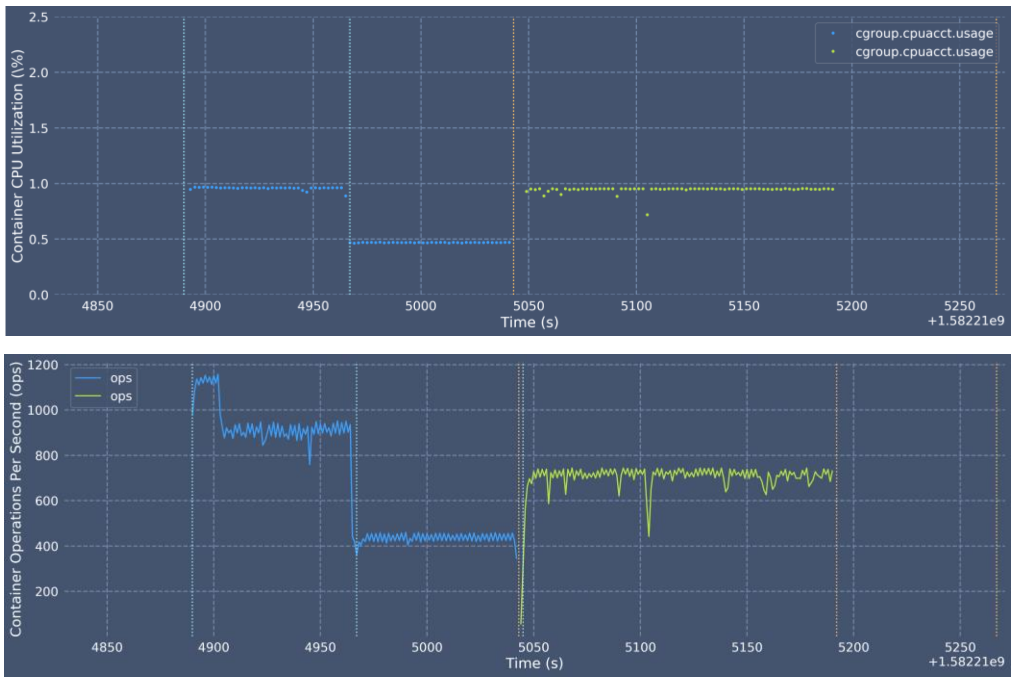

Figure 5-11 Container CPU Utilization (top), same container workload's instant performance metrics (bottom)

5.3.2 CPU workload container placement policy

Our initial placement policy used a greedy approach that tries to manage resource provisioning by live migration containers from a Compute host with high CPU utilization to a host in the system that has the most CPU resources available. This policy normalizes container CPU utilization based on host core count and picks the container with the lowest CPU usage and moves it to the target host. This placement policy helped us develop Herd Control and understand the performance of the whole Herd system once integrated. We named this policy the *CPU Greedy Policy*.

To test this placement policy, we developed a new type of scenario that is randomly generated based on a set of parameters. These include total experiment length, individual container workload length, and other parameters such as workload thread count, memory utilization and total number of Compute hosts available. To generate these scenarios a GoLang application was developed by one of our undergrads research students. It used a random seed that can regenerate the same output, it is based on our *staircase* scenario scripts. Each randomly generated scenario launches a total of around 400 containers placed on Compute hosts at different time intervals.

To run these scenarios, we deployed a new testbed on an Openstack cluster. The cluster consisted of two Dell PowerEdge R6515 with AMD EPYC processors with 64 vCores and 128GB of RAM each. This testbed was used to run from 2 to 30 virtual machines, depending on the workloads, that acted as Compute Nodes. We automated the deployment of different Openstack testbeds and Compute node virtual machines to easily change system configuration.

To evaluate our policy, we ran the randomly generated scenarios without Herd Control acting upon the system; this is our performance baseline and control experiment. The instant performance metrics for each container were recorded to validate the experimental results (Table 5-1).

	Control	CPU Greedy Policy	Gain vs Control
Average Hosts CPU utilization	0.68	0.80	17.94%
Average Containers CPU utilization	0.73	0.80	10.69%
Average Workloads Ops	579.85	617.03	6.41%

Table 5-1 Greedy Policy performance

We also developed a cheating policy that understands the workload and tires to set an ideal container placement. We use this policy to compare our *CPU Greedy Policy* with the baseline (control) experiment to see how far off we are from an *ideal* system state (Table 5-2).

	Control	Ideal	CPU Greedy Policy	Gain vs Control
Average Hosts CPU utilization	0.46	0.66	0.55	19.49%
Average Containers CPU utilization	1.23	1.38	1.33	7.74%
Average Workloads Ops	2,025.04	2,328.41	2,173.40	7.33%

Table 5-2 Greedy Policy performance compared to ideal case

The CPU Greedy Policy has a baseline of 1 for the number of processes running within the container. Thanks to our CPU workload characterization experiments we learned that the Compute host scheduling algorithm equally shares CPU resources, regardless of the amount of process within a container, with all containers. Our characterization experiments showed us that a multi-threaded workload will be the most affected by other container's resource consumption. The cheating policy knows the number of threads withing each container as to place them in an ideal host. The closest metric that could be monitored, that resembles the number of threads of a workload in a container, is the process identifier (PID) count of a container. This metric is not available in PCP, or Docker Stats or its API. We obtain this metric from directly from CGROUP filesystem in the compute hosts and stream it to Herd Monitor. With this new information we developed a new placement policy that handles multi-threaded workloads (Table 5-3).

	Control	**Ideal**	**CPU Greedy Policy for Multi-thread Containers**	**Gain vs Control**
Average Hosts CPU utilization	0.46	0.66	0.59	28.67%
Average Containers CPU utilization	1.23	1.38	1.43	16.56%
Average Workloads Ops	2,025.04	2,328.41	2,336.46	15.38%

Table 5-3 Improved Greedy Policy performance

Our container placement policies show that our Herd resource provisioning platform for container can improve overall system performance of the running workloads.

See **Appendix III - Tests list** for a complete list of all the tests, experiments and workload characterization tests we ran during the development of Herd.

5.4 Machine Learning

For feature selection and performance forecasting, we used the gradient boosting implementation XGBoost [44]. Gradient boosting is a machine learning technique for regression problems. Used for estimating relationships between an outcome variable and features. Gradient boosting works by producing a group of weak prediction models that can be used together to form a strong prediction model (Figure 5-12).

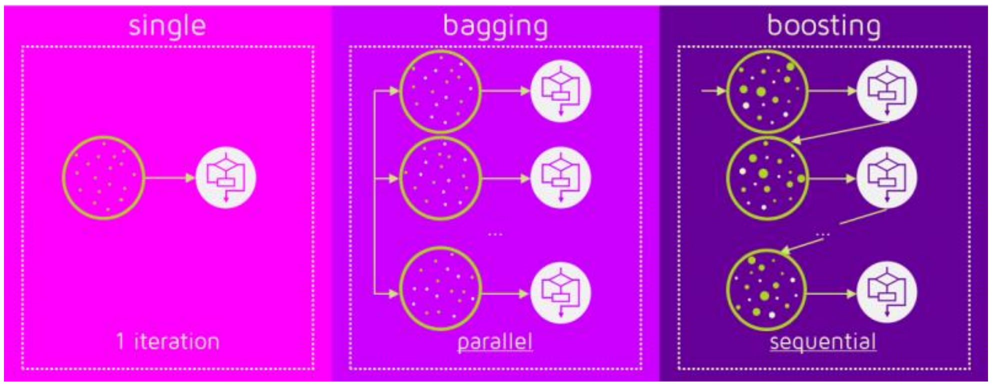

Figure 5-12 Bagging (independent models) & Boosting (sequential models).

"In Boosting algorithms each classifier is trained on data, taking into account the previous classifier's success. After each training step, the weights are redistributed. Misclassified data increases its weights to emphasize the most difficult cases. In this way, subsequent learners will focus on them during their training."[45]

We selected this machine learning model and implementation as it proved, based on our literature review, that it can provide relatively accurate forecasting for its short training time. We tested metric forecasting on CPU and GPU using the Azure Resource Central's dataset for VM CPU utilization. XGboost provides multiple approximation algorithms with various degrees of accuracy and training speed. We found that its GPU and CPU performance is similar for the *hists* tree construction algorithm that is an optimized approximate greedy algorithm.

With the use of our monitored resource utilization data and workload validation output, in the form of operations per seconds (*ops*), we can train a machine learning model to forecast container performance. By utilizing resource metrics, we can forecast the performance of a container when running on another host. Our dependent variable is the container *ops*, our independent variable are the resource metrics of the container itself and its host. We forecast a container's performance by using the resource metrics of a container and those of a target host. We predict the operations per second of a container on said target host.

Due to the ascynchronic nature of the monitoring daemon, to forecast a container we need to re-sample the metrics to a given sample window. By re-sampling the metrics, we can align them in time to have a direct comparison of a container's resource utilization in regard to its host.

Our methodology was to run many randomly generated scenarios, gather all the data, resample the metrics to a sample window, close or equal to the monitor sampling rate. We then joined the container metrics with its host metrics and workload operations per second using *dataframes*. We trained a XGBoost regressor by splitting the data into a training set and a testing set. We forecasted the operations per second for all containers by evaluating the model with host and container metrics (Figure 5-13). We also tested single container performance forecasting to validate the model (Figure 5-14). We evaluated the model for a container by providing data for another host that isn't the container's current compute host.

We conducted a preliminary evaluation of online transaction processing workloads using Sysbench. We deployed a MySQL container that acts as the OLTP server and a Sysbench container that acts as the OLTP client. The initialization process requires the building and population of a database on the OLTP server container by the OLPT client container. Once the database is created, the workload begins. The Sysbench OLTP Client executes SQL queries to the OLTP server container. For this setup we used network bound container live migration with filesystem migration support. Sysbench OLTP tests also provide instant performance metrics that were used to evaluate our machine learning model for performance forecast and feature selection.

Below are the parameters and percentage of error of the forecasters for each workload type (Table 5-4).

Workload	**Samples**	**Regressor estimators**	**Early Stopping rounds**	**Mean absolute error**	**Mean absolute percentage error**
CPU	10475	1000	100	36.02	4.13
Memory	23871	1000	100	320.32	7.83
OLTP	1493	1000	100	7.31	2.68

Table 5-4 XGBoost parameters and results

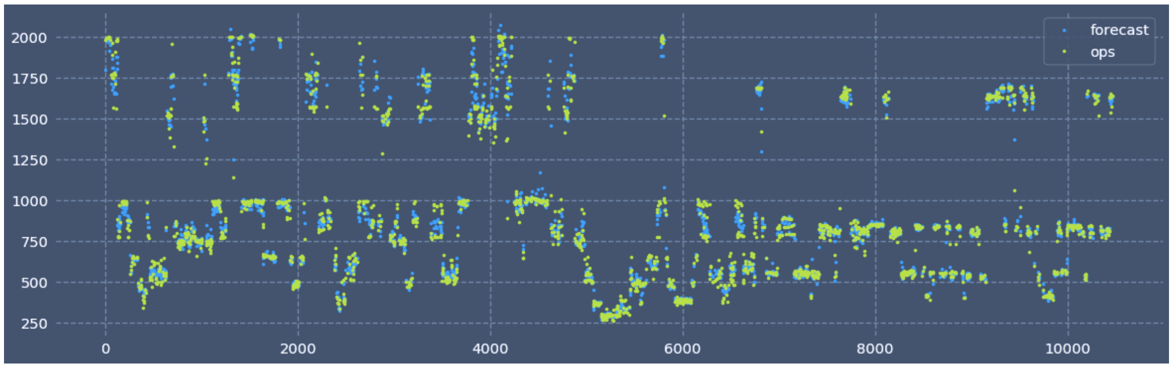

Figure 5-13 Forecast of a CPU Workload's performance over time for all containers on a scenario vs actual operations per second

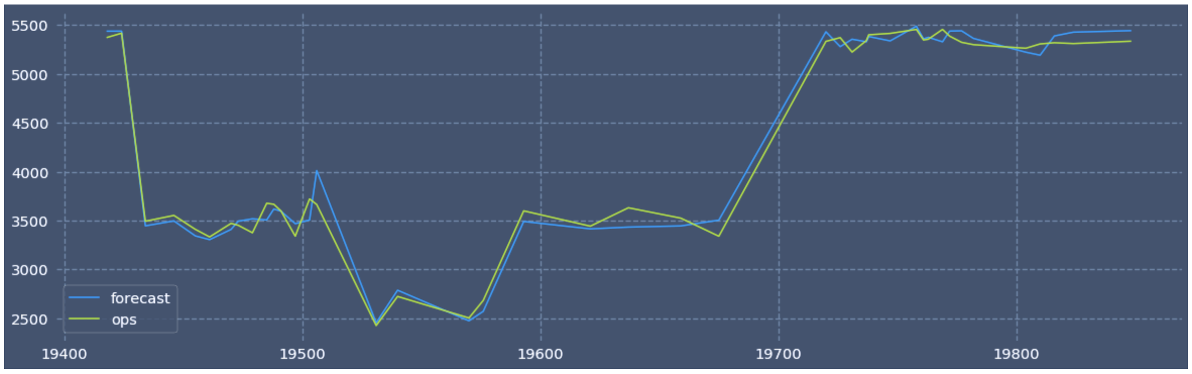

Figure 5-14 Forecast of a Memory Workload’s performance over time for a single container vs actual operations per second

We also used XGBoost for feature selection. To understand the performance metrics of an OLTP workload, we used the feature importance plot for a trained gradient boosting model using host and container data for OLPT scenarios. We identified disk IO operations, disk activity and CPU utilization as the main metrics that can be used as a performance indicator (Figure 5-15).

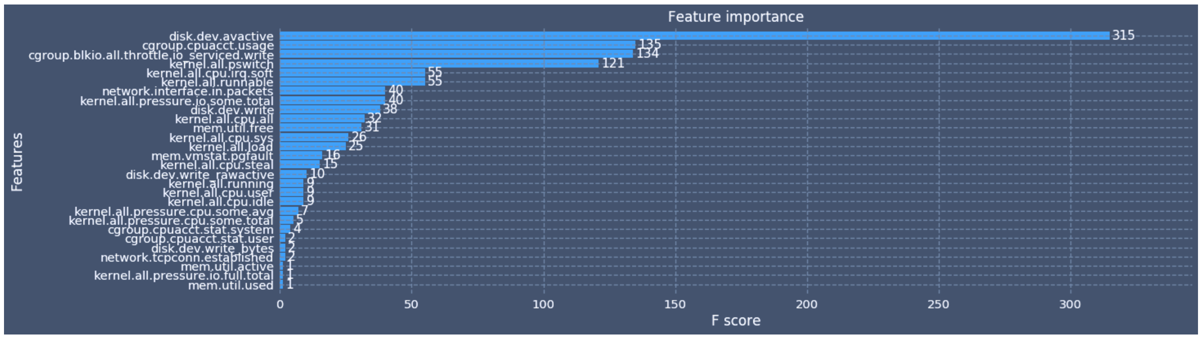

Figure 5-15 Feature importance plot for a OLTP workload

5.5 OLTP Policy

With the information gathered through XGBoost's feature selection and the characterization experiments for the OLTP workload, we developed a preliminary policy to handle OLTP container resource provisioning. We focused our policy on compute host disk activity and container disk access requests. This is a greedy policy that takes the container with the highest disk request per seconds metric from the compute host with the highest percentage of disk activity. The container is live migrated to the Compute host with the lowest disk activity percentage. The preliminary OLTP policy allowed us to better understand the OLTP server's performance while live migrating by evaluating resource metrics with performance metrics (Figure 5-16). We found out that a policy based on disk activity is not ideal as even a small amount of disk request generates a high percentage of disk activity.

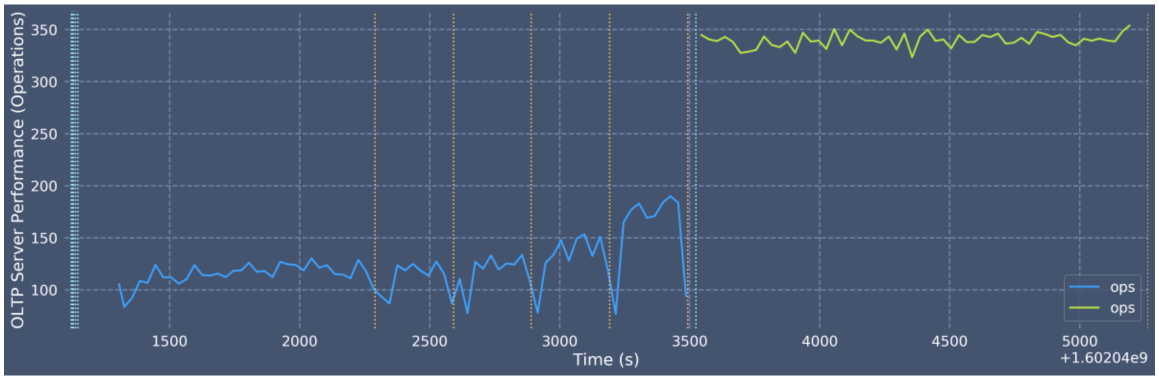

Figure 5-16 OLTP Server container performance gains due to live migration

5.5.1 TPC-C

TPC-C is the benchmark published by the Transaction Processing Performance Council [46] (TPC) for Online Transaction Processing (OLTP). It runs a mix of five concurrent transactions of different types and complexity either executed on-line or queued for deferred execution. The TPC-C database is comprised of nine types of tables with a wide range of record and population sizes that portrays the activity of a wholesale supplier. Its performance is measured in transactions per minute (tpmC).

TPC-C simulates a complete environment where a population of terminal operators executes transactions against a database. The benchmark is centered around the principal activities (transactions) of an order-entry environment. In the TPC-C business model, a wholesale parts supplier (called the Company below) operates out of a number of warehouses and their associated sales districts. The TPC benchmark is designed to scale just as the Company expands and new warehouses are created.

We used TPC-C workloads to evaluate the performance of our container placement policies. The TPC-C implementation we used outputs an overall benchmark score at the end of the execution and does not provide instant performance metrics. We performed a workload characterization to understand the behavior of the workload in and tested our OLTP container placement policy with TPC-C workloads.

The workload is very similar to Sysbench OLTP workload. It runs a database sever container and a client container. The workload parameters allow scaling of connected clients to the database server and the size of the database. We identified similar behavior to the OLTP workload such as heavy disk IO (Figure 5-17).

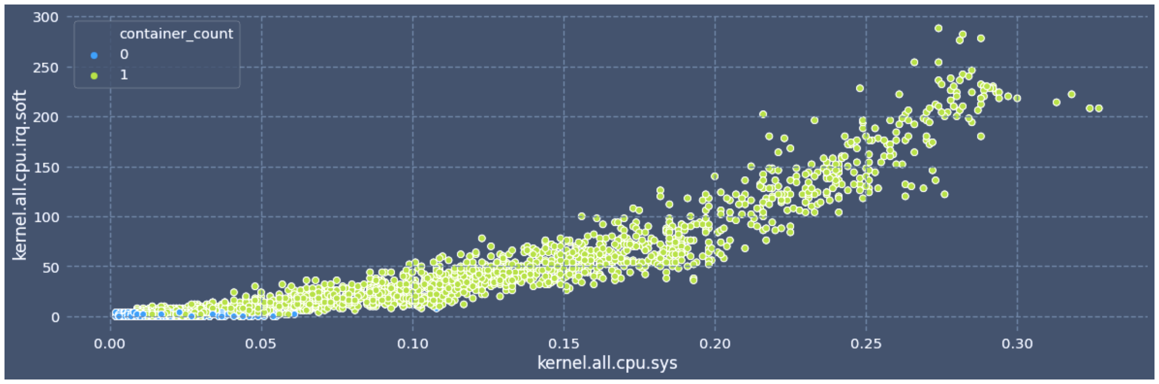

Figure 5-17 Host's Interrupts vs Kernel CPU utilization while running an TPC-C server container

We conducted a scaling test by varying the number of TPC-C terminals (clients) connected to a server to understand system limits and tune the database configuration accordingly.

We deployed a 6-Compute node and 1 Control node testbed in Openstack for network-bound container live migration with filesystem support (Figure 5-18). Our scenarios consisted on launching 5 TPC-C servers with a database of 10 Warehouses. On another compute host we launched one TPC-C client container per each TPC-C server. Each client container simulated 10 Terminals per warehouse for a total of 100 terminals connected to teach server. The ideal result is that Herd Control instructs the migration of TPC-C servers to other Compute hosts for better resource provisioning.

We developed a new OLTP migration policy based on disk requests per compute host and container. To mitigate problems, such as container bouncing, we only evaluate hosts and containers that reach a minimum threshold of monitored samples. We set resource reward thresholds for disk activity and disk request to be reached for a host to be a valid target Compute host for live migration. Based on our workload characterization, we set a disk request baseline that is used for the disk request's threshold. This baseline is experimentally computed. We ran an IO Sysbench workload and monitored it with Herd Monitor to compute said baseline.

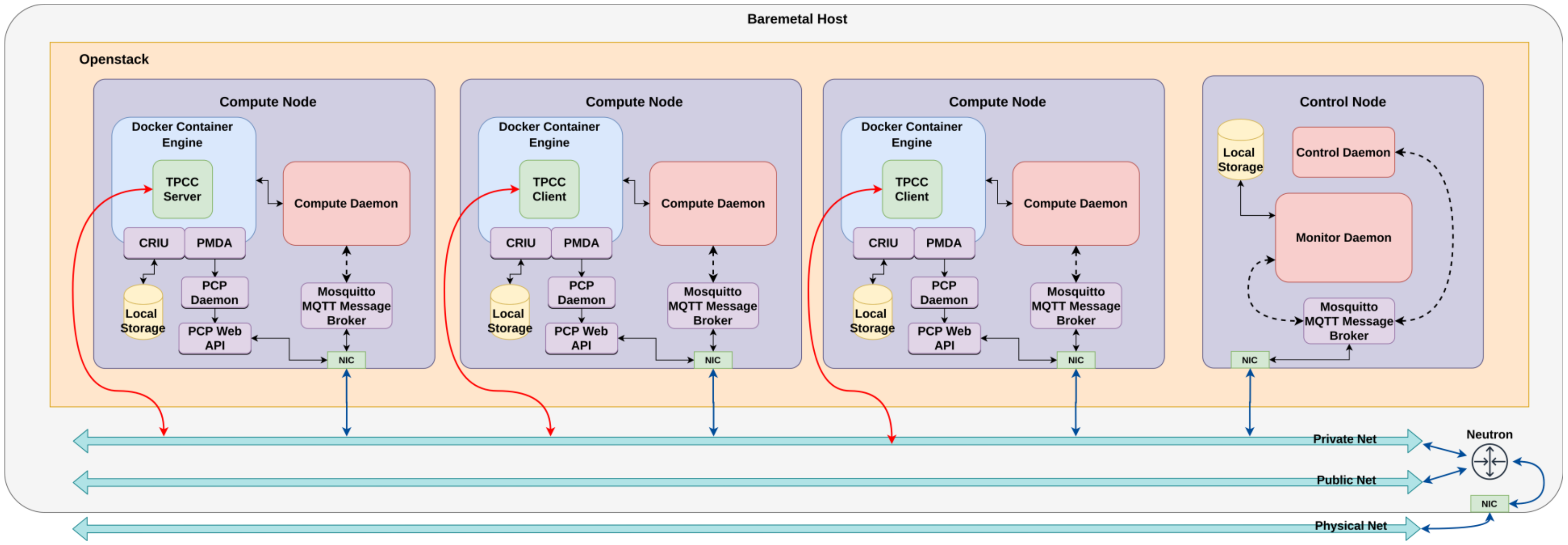

Figure 5-18 Diagram of an Openstack testbed for network bound container live migration showing 3 compute hosts and a control host.

5.5.2 OLTP Policy for TPC-C Workloads

Our OLTP Policy finds the container with the highest disk ops and set as candidate, migrate the candidate container to the compute host with minimum disk ops if:

- Candidate container has enough resource metric samples.
- Compute hosts with the minimum disk ops compared to the compute host with maximum disk ops have reached the disk ops reward threshold.
- Compute host with the highest disk activity has reached the disk active threshold.
- Compute hosts with the minimum disk activity compared to the compute host with maximum disk activity have reached the disk activity reward threshold.
- Candidate container has reached the disk ops baseline threshold.
- Compute host with the minimum disk ops is not the same as the candidate container's current compute host.

If those conditions are not met, re-evaluate the system and search for new candidate containers, if no new candidates are found evaluate the system with the *CPU Greedy Policy for Multi-thread Containers.*

We tested our policy with the following parameters:

Disk Ops Baseline = 8000
Disk Ops Baseline Threshold = 0.005
Disk Ops Reward Threshold = 0.35
Disk Active Threshold = 0.15
Disk Active Reward Threshold = 0.15
Missing Samples Threshold = 0.02
Herd Monitor Sampling Rate = 5s
Herd Monitor Container Tracking Rate = 2s
Herd Control Sample Window= 30s
Herd Control Evaluation Rate= 150s
Herd Control Initial Evaluation Delay = 150s

Our OLTP policy showed that CPU load lowers as containers are move out of a Compute node (Figure 5-19). It also shows how disk throughput lows when containers are moved out of the Compute node (Figure 5-20) and how a container's disk throughput increases as it moves to other Compute nodes (Figure 5-21)

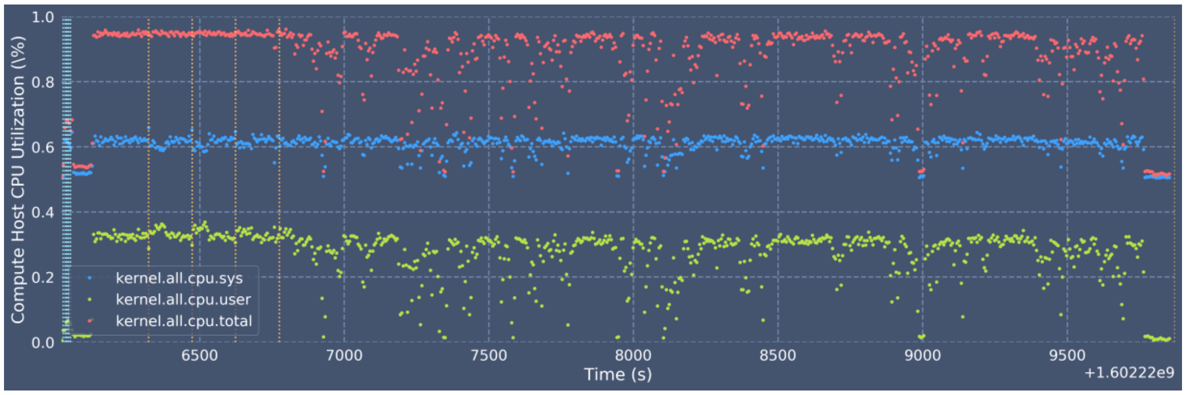

Figure 5-19 Host CPU utilization as TPC-C Server containers are migrated out (orange line are migrations)

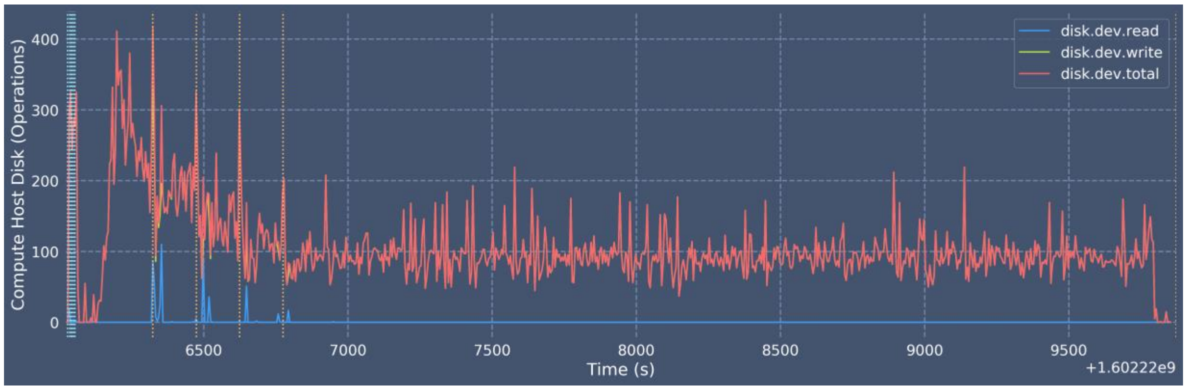

Figure 5-20 Host Disk request as TPC-C Server containers are migrated out (orange line are migrations)

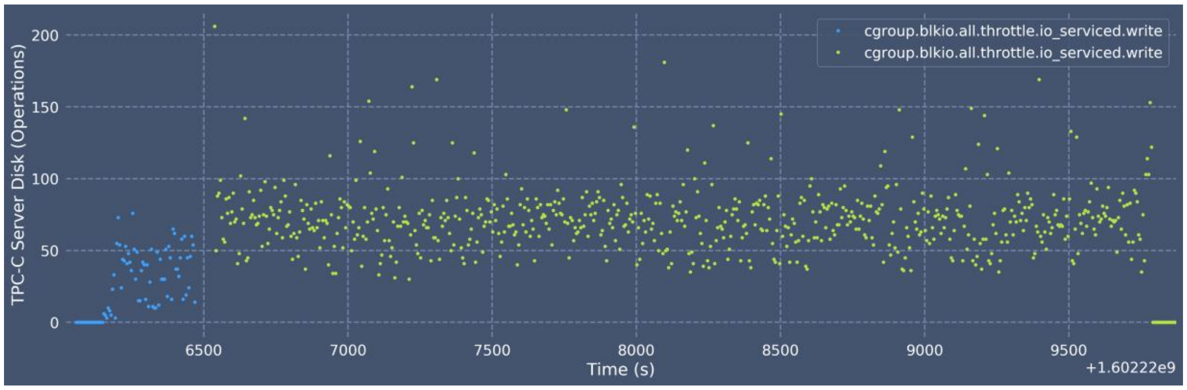

Figure 5-21 TPC-C Server Container Disk request gains as it is moved to a new host (blue old host, green new host)

To evaluate the policy, we ran a base experiment without migrations on a 2-core CPU compute host and on a 4-core CPU compute host. We then ran the same experiments using our placement policy. We ran each experiment 10 times for a total of 40 experiments. Our final results show an increase in tractions per minute 49.05 vs 110.17 tpcM for a 2 core compute hosts and 51.51 vs 112.01 tpcM for a 4 core Compute node (Figure 5-22).

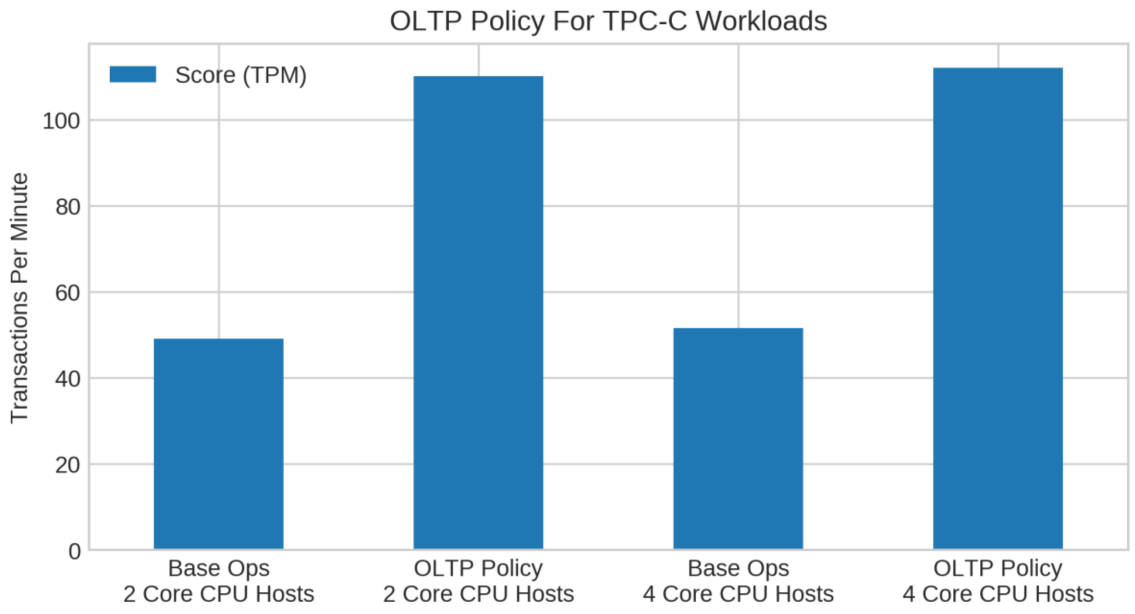

Figure 5-22 OLTP Policy for TPC-C workloads overall performance score

5.5.3 Discussion

The performance benefits of the policy on our TPC-C workload implementation are evident on the total amount of transactions per minutes. The overall performance will be affected the experiment's duration and the evaluation period of the policy. A faster evaluation will migrate containers quicker and give better overall score. The duration of the workload will affect its performance, as the more time a server runs on a new host, more total transaction can be performed with better performance. We conduced warehouse scaling test to understand the systems limits (**Figure 5-23**). It helps us understand how the TPC-C workload behaves with an increasing number of Warehouses. We found out that once the system reaches memory limits the transaction per minute stops scaling and the system fails (Figure 5-24).

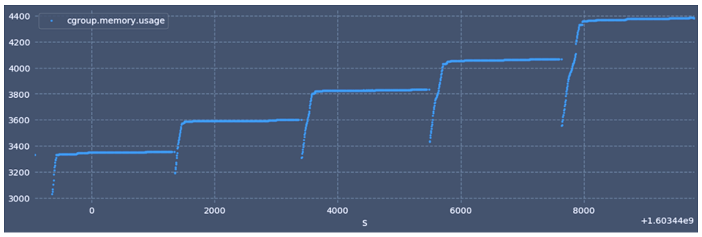

Figure 5-23 Memory usage overtime of a TPC-C server running 1,2,3,4 and 5 warehouse workloads with 10 terminals per warehouse each.

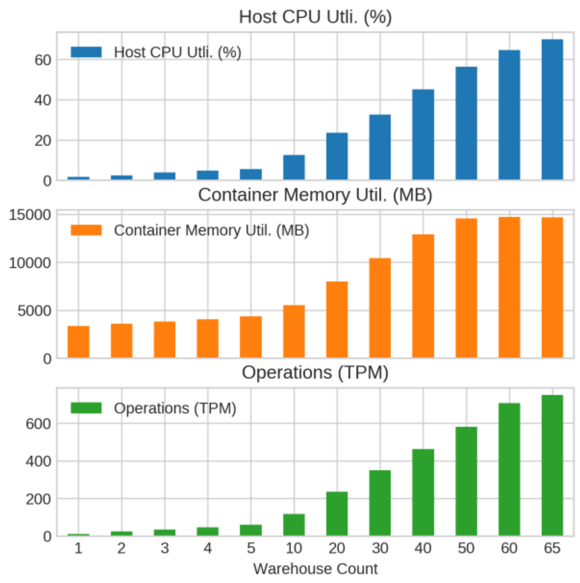

Figure 5-24 TPC-C Warehouse scaling test

We also conducted staircase scenarios to test Disk scaling. Due to the nature of our virtualized testbed, live migration might not produce positive benefits as all compute hosts are accessing the same physical device. Our staircase test shows that the TPC-C servers does not CPU limits on an 8 core CPU compute host (Figure 5-25) and reaches a reach disk limits that we use as parameter to calibrate our policy (Figure 5-26).

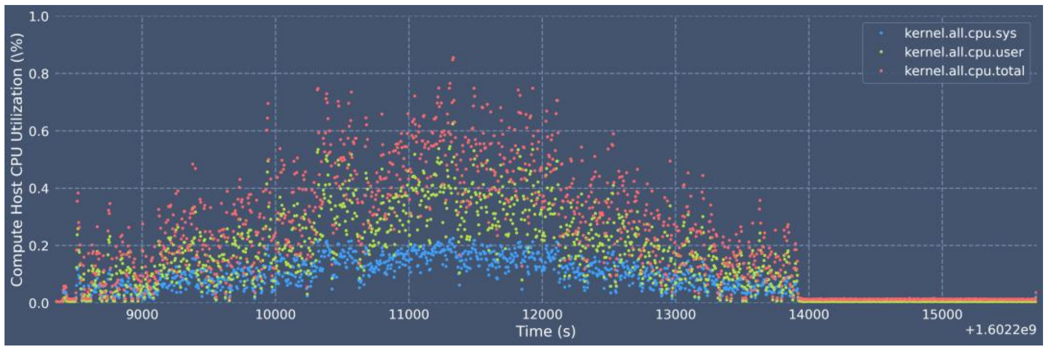

Figure 5-25 Compute host CPU utilization on a TPC-C staircase scenario.

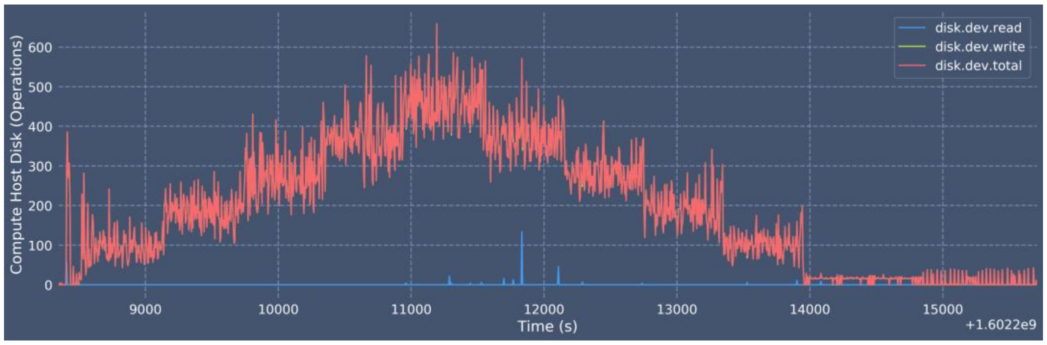

Figure 5-26 Compute host Disk request on a TPC-C staircase scenario.

To test our hypothesis, that the baremetal host is the limiting factor in our test, we monitored the meta-host during one of our policy evaluation experiments. When running on an 8-core CPU compute host, the disk request is affected by live migration (Figure 5-27), but the meta-host disk request stays the same (Figure 5-28). The experiments resulted in no container performance benefit with live migration. We conducted more experiments varying the amount of CPU cores on the compute host. Our experiments showed that for our specific testbed, a 4 core or 2 core compute hosts can be used for resource provisioning (Figure 5-29). The Virtual Machine's Kernel CPU utilization reaches system limits, bottle necking resource access to disk for requests. We used this to our advantage to synthetically provision disk access to better evaluate our experiments**.**

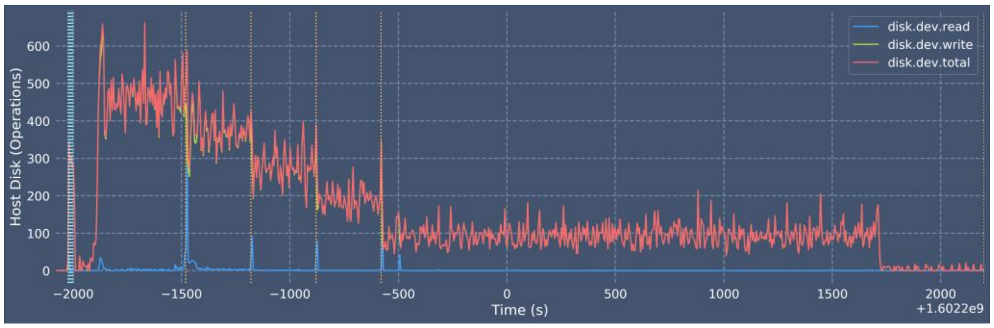

Figure 5-27 8-core CPU Compute host's disk request while migrating TPC-C containers out

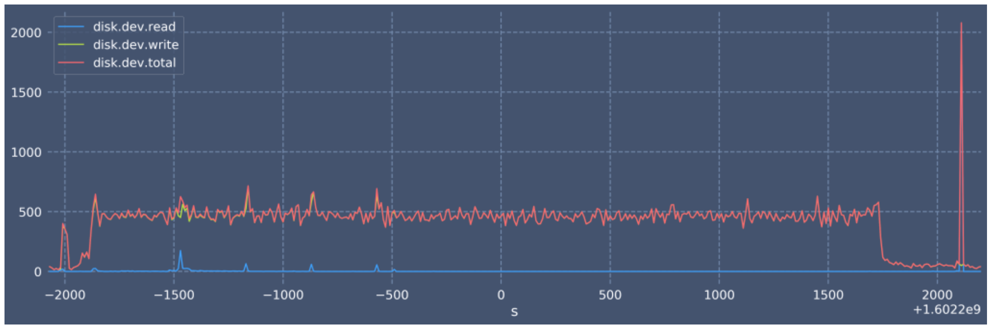

Figure 5-28 Baremteal's (meta-host) disk request running a 8 core CPU compute host while that compute host is migrating TPC-C containers out

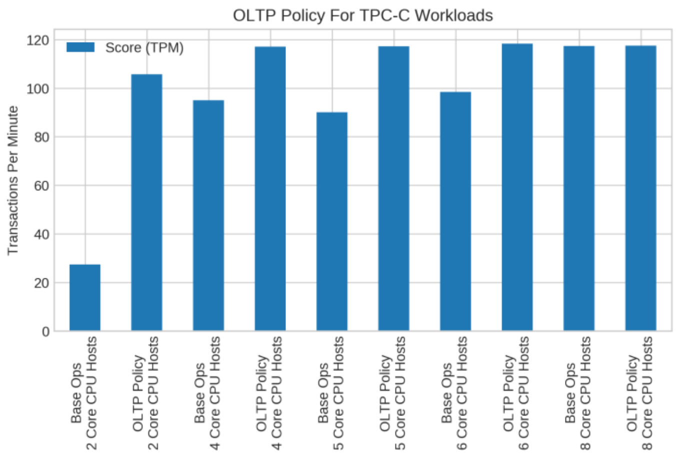

Figure 5-29 TPC-C overall score for compute hosts with different CPU core-counts

6 CONCLUSION

We developed an optimized container live migration toolset that can quickly migrate network-bound containers with filesystem support. We designed and developed a lightweight resource monitoring tool that supports live-migrating containers. We developed a resource provision toolset that implements different container placement policies. We compared the performance of different live migration techniques and developed our own optimized techniques. We developed a methodology to process system and container resource information for machine learning purposed. We designed a machine learning model for container resource provision and forecasting. We compared different provisioning algorithms. We design, developed, implement, tested and characterized a Distributed Resource Provisioning for Containers Using Machine Learning and Live Migration.

As future work we can continue evaluation different migration policies. We can perform more system optimizations and handle support for other real-time streaming platforms and container orchestration software. We can continue testing our system for other uses cases and platforms, such as IoT systems. Finishing porting our toolset to handle ARM computer architectures and testing for other computer architectures such as RISC-V. Filesystem optimization can potentially open another area of research to handle better support for container live migration. We plan on publishing our toolset components and large monitoring dataset as an open-source project.

REFERENCES

[1] S. Hogg, "Software Containers: Used More Frequently than Most Realize | Network World," *Network World*, 2014. https://www.networkworld.com/article/2226996/software-containers--used-more-frequently-than-most-realize.html (accessed Oct. 28, 2020).

[2] Docker Team, "What is a Container? | App Containerization | Docker," *Docker*. https://www.docker.com/resources/what-container (accessed Oct. 28, 2020).

[3] IBM Cloud Team, "Containers vs. VMs: What's the Difference? | IBM," *IBM*, Sep. 02, 2020. https://www.ibm.com/cloud/blog/containers-vs-vms (accessed Oct. 28, 2020).

[4] "CRIU," *criu.org*, Apr. 29, 2020. https://criu.org/Main_Page (accessed Oct. 28, 2020).

[5] "Horizontal Pod Autoscaler | Kubernetes," *Kubernetes*. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#algorithm-details (accessed Oct. 28, 2020).

[6] "Slurm Workload Manager," *Slurm*, Nov. 24, 2013. https://slurm.schedmd.com/slurm.html (accessed Oct. 28, 2020).

[7] G. Berg and M. Brattlöf, "Distributed Checkpointing with Docker Containers in High Performance Computing," Sweden, Jun. 2017. Accessed: Oct. 28, 2020. [Online]. Available: www.hv.se.

[8] M. Kudinova and P. Emelyanov, "Building mathematical model for restoring processes tree during container live migration," in *Proceedings - 2017 4th International Conference on Engineering and Telecommunication, En and T 2017*, Dec. 2017, vol. 2017-Janua, pp. 160–164, doi: 10.1109/ICEnT.2017.41.

[9] X. Merino Aguilera, C. Otero, M. Ridley, and D. Elliott, "Managed Containers: A Framework for Resilient Containerized Mission Critical Systems," in *IEEE International Conference on Cloud Computing, CLOUD*, Sep. 2018, vol. 2018-July, pp. 946–949, doi: 10.1109/CLOUD.2018.00142.

[10] J. Lee and K. Kang, "Poster: A lightweight live migration platform with container-based virtualization for system resilience," in *MobiSys 2017 - Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, Jun. 2017, p. 158, doi: 10.1145/3081333.3089302.

[11] E. Arzuaga and D. R. Kaeli, "Quantifying load imbalance on virtualized enterprise servers," in *WOSP/SIPEW'10 - Proceedings of the 1st Joint WOSP/SIPEW International Conference on Performance Engineering*, 2010, pp. 235–242, doi: 10.1145/1712605.1712641.

[12] C. O. Rolim, F. Schubert, A. G. M. Rossetto, V. R. Q. Leithardt, C. F. R. Geyer, and C. B. Westphall, "Comparison of a Multi output Adaptative Neuro-Fuzzy Inference System (MANFIS) and Multi Layer Perceptron (MLP) in Cloud Computing Provisioning."

[13] A. Biswas, S. Majumdar, B. Nandy, and A. El-Haraki, "Automatic resource provisioning: A machine learning based proactive approach," in *Proceedings of the International Conference on Cloud Computing Technology and Science,*

CloudCom, Feb. 2015, vol. 2015-Febru, no. February, pp. 168–173, doi: 10.1109/CloudCom.2014.147.

[14] M. Cheng, J. Li, and S. Nazarian, “DRL-cloud: Deep reinforcement learning-based resource provisioning and task scheduling for cloud service providers,” in *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*, Feb. 2018, vol. 2018-Janua, pp. 129–134, doi: 10.1109/ASPDAC.2018.8297294.

[15] E. Cortez, M. Russinovich, A. Bonde, M. Fontoura, A. Muzio, and R. Bianchini, “Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms?,” in *SOSP 2017 - Proceedings of the 26th ACM Symposium on Operating Systems Principles*, 2017, pp. 153–167, doi: 10.1145/3132747.3132772.

[16] A. Ahmed and G. Pierre, “Docker-Pi: Docker container deployment in fog computing infrastructures,” *IEEE Int. Conf. Mob. Cloud Comput.*, vol. 9, no. 1, pp. 6–27, Apr. 2020, doi: 10.1504/IJCC.2020.105885.

[17] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, “Sandpiper: Black-box and Gray-box Resource Management for Virtual Machines,” Amherst, 2009.

[18] C. Mergenci and I. Korpeoglu, “Generic resource allocation metrics and methods for heterogeneous cloud infrastructures,” *J. Netw. Comput. Appl.*, vol. 146, p. 102413, 2019, doi: 10.1016/j.jnca.2019.102413.

[19] P. Souza Junior, D. Miorandi, G. Pierre, and G. Pierre Stateful, “Stateful Container Migration in Geo-Distributed Environments,” 2020. Accessed: Nov. 04, 2020. [Online]. Available: https://hal.inria.fr/hal-02963913.

[20] “(No Title).” https://podman.io/getting-started/ (accessed Nov. 04, 2020).

[21] “Understanding the Docker Internals | by Nitin AGARWAL | Medium.” https://medium.com/@BeNitinAgarwal/understanding-the-docker-internals-7ccb052ce9fe (accessed Nov. 04, 2020).

[22] “What cannot be checkpointed - CRIU.” https://www.criu.org/What_cannot_be_checkpointed (accessed Nov. 04, 2020).

[23] “MQTT - The Standard for IoT Messaging,” *MQTT*. https://mqtt.org/ (accessed Oct. 28, 2020).

[24] “What is OLTP? | IBM.” https://www.ibm.com/cloud/learn/oltp (accessed Nov. 04, 2020).

[25] A. Reber and M. Rapoport, “Container Migration All Around The World,” no. 688386, 2017, Accessed: Oct. 29, 2020. [Online]. Available: http://xonotic.org/.

[26] Docker Inc., “Use Bridge Networks - Docker Documentation,” 2019. https://docs.docker.com/network/bridge/ (accessed Oct. 29, 2020).

[27] Docker Inc., “Use Overlay networking - Docker Documentation,” 2019. https://docs.docker.com/network/overlay/ (accessed Oct. 29, 2020).

[28] Nomad by HashiCorp, “Nomad by HashiCorp,” 2019. https://www.nomadproject.io/use-cases/automated-service-networking-with-consul (accessed Oct. 29, 2020).

[29] Docker Inc., “Use Host networking - Docker Documentation,” 2019.

https://docs.docker.com/network/host/ (accessed Oct. 29, 2020).
[30] "MacVTap - Linux Virtualization Wiki," Dec. 30, 2017. https://virt.kernelnewbies.org/MacVTap (accessed Oct. 29, 2020).
[31] "keepalived.conf(5) — keepalived — Debian unstable — Debian Manpages." https://manpages.debian.org/unstable/keepalived/keepalived.conf.5.en.html (accessed Nov. 04, 2020).
[32] "Introduction to Linux interfaces for virtual networking - Red Hat Developer." https://developers.redhat.com/blog/2018/10/22/introduction-to-linux-interfaces-for-virtual-networking/ (accessed Nov. 04, 2020).
[33] "Live migration - CRIU," *criu.org*, Jan. 13, 2019. https://criu.org/Live_migration (accessed Oct. 28, 2020).
[34] "Exporters and integrations | Prometheus," *Prometheus*. https://prometheus.io/docs/instrumenting/exporters/ (accessed Oct. 28, 2020).
[35] "Overview - Vector," *Vector*. https://getvector.io/docs/ (accessed Oct. 28, 2020).
[36] Prometheus Authors 2014-2020, "Grafana | Prometheus," *Prometheus*. https://prometheus.io/docs/visualization/grafana/ (accessed Oct. 28, 2020).
[37] "Performance Co-Pilot Grafana Plugin — grafana-pcp 3.0.0-beta2 documentation," *Grafana*. https://grafana-pcp.readthedocs.io/en/latest/index.html (accessed Oct. 28, 2020).
[38] "Performance Co-Pilot." https://pcp.io/docs/guide.html (accessed Nov. 04, 2020).
[39] "Performance Co-Pilot." https://pcp.io/man/man3/pmda.3.html (accessed Nov. 04, 2020).
[40] "Docker - Cannot remove dead container - Stack Overflow." https://stackoverflow.com/questions/30794108/docker-cannot-remove-dead-container (accessed Nov. 04, 2020).
[41] "Use the OverlayFS storage driver | Docker Documentation." https://docs.docker.com/storage/storagedriver/overlayfs-driver/ (accessed Nov. 04, 2020).
[42] "Last Will and Testament - MQTT Essentials: Part 9." https://www.hivemq.com/blog/mqtt-essentials-part-9-last-will-and-testament/ (accessed Nov. 04, 2020).
[43] R. Light, "MQTT | Eclipse Mosquitto," *Mosquitto*. https://mosquitto.org/man/mqtt-7.html (accessed Oct. 28, 2020).
[44] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Aug. 2016, vol. 13-17-August-2016, pp. 785–794, doi: 10.1145/2939672.2939785.
[45] "What is the difference between Bagging and Boosting? | Quantdare." https://quantdare.com/what-is-the-difference-between-bagging-and-boosting/ (accessed Nov. 04, 2020).
[46] "TPC-C Overview." http://www.tpc.org/tpcc/detail5.asp (accessed Nov. 04, 2020).

APPENDIX I – HERD MONITOR PSEUDOCODE

```
get cli parameters
set sampling rate
set monitoring timestamp
get hostlist
build cvs list, initiate logs


instantiate globally shared objects (multi host)


start MsgQueue thread


for each host in hostlist do
          instantiate shared objects (same host)


          create pcp context for host


          start Host Monitoring Thread
          start Container Monitoring Thread
          start Migration Tracking Thread
          start Remote Logger Thread


done
```

Daemon thread pseudocode

```
get metric configurations from config file
compute time delta from previous run


for each metric in config file do:


          get datatype
          get operations to apply to current metric
          for each operation do:
                    apply operation based on datatype and time delta
                    clean output
                    store to metric list
          done
done
```

Metrics Processor thread pseudocode

```
every samplerate seconds do


          from hostconfig generate pcp metric to request string
          get pcp context for current host
          poll pmwebd server with request string


          parse response to json
          for every received metric do
                              get metric name
                              for every instance in metric
                                        get instance to measure id from hostconfig
                                        add correct instance value and name to metric list
                              done
          done


          process metric list with Metric Processor
          write processed metrics to cvs
          publish processed metrics to mqtt
done
```

Host Monitor thread pseudocode

```
every trackingSamplerate seconds run:

          get pcp context for current host
          poll pmwebd server for all container names and their id
          poll pmwebd server for active cgroups ids

          filter all containers names by active cgroups ids
          generate active container list and compare with previous run

          compute removed containers and add them to action queue
          compute added containers and add them to action queue

          write migration metric to cvs
          publish migration metric to mqtt

done
```

Migration Tracker thread pseudocode

```
every samplerate seconds run:

          build active container list from container action queue

          for every active container do

                    from containerconfig generate pcp metric to request string
                    get pcp context for current host
                    poll pmwebd server with request string

                    parse response to json
                    for every received metric do
                              get metric name
                              for every instance in metric
                                        get instance-to-measure id from containerconfig
                                        add correct instance value and name to metric list
                              done
                    done

                    process metric list with Metric Processor
                    write processed metrics to cvs
                    publish processed metrics to mqtt

          done
done
```

Container Monitor thread pseudocode

APPENDIX II – HERD MONITOR TECHNICAL CHALLENGES

The main feature of Herd Monitor, that differentiates it from other container monitoring tools, is its ability to handle containers live migrating and its resource metrics. During the development of the Migration Tracking features we encountered problems with the way Docker and PCP handled CGROUPs.

Migration tracking errors were rare but its symptoms became apparent on the desynchronization of the added/removed containers per-host queues used by the Monitoring thread generated by the Tracking thread. Desynchronization in this case means the removal of a (marked-as) non-existing container or the addition of an (marked-as) existing container. Although monitoring errors can be somewhat tolerated, on long scenarios, this is an issue. Queue desynchronization was evidently the symptom, finding out the source of the problem was not easy. A concurrency issue was ruled out as the queues are thread-safe and concurrency was already exhaustively tested. An inter-host race condition (eg. adding a container to another host before removing it from the previous host) was ruled out, as an active container is marked as such, only when it's "running" and migration stops the container. Herd Compute makes sure an instance of a container is only running in one host even in the case of a rollback. The severity of the problem was apparent when the system was tested on a new host, as the errors became constant. Fixing the problem required an exhaustive test of all possible tracking errors.

A bug in Docker[40], where a container wasn't reported by Docker and could not be removed but was present on the PCP CGROUP metrics, created a tracking mismatch (CGROUP vs container names) on a Compute Hosts. The initial Migration Tracking code was written in a way that once the invisible containers where removed the Tracking broke. To remove those container a AUFS Docker filesystem reset was made and the system was set to use Overlay2[41] filesystem.

Fixing the error required not only fixing the original bug (tracking mismatch) but also a complete rewrite of the main function of the Tracking thread by filtering all instances of containers by their status. The problem was complex as the point of the Tracking thread is to concurrently keep track of container migration count and monitoring (to improve performance) by finding out, inter-host, not only added/removed containers but also re-added containers. As a result, system performance was not be negatively affected, but rather improved. In the case of error, a correctly synchronized tracking no longer yielded monitoring for non-existing containers, which caused overhead on the Compute hosts and errors in Herd Monitor.

APPENDIX III - TESTS LIST

```
Herd Experiment Log

Folder Data/

01 pong-random migrations mixed workload tests, and monitoring tests
init herdmon tests
correlation chart

02-04 pong-controlled migrations mixed workload tests, and monitoring tests

05-07 scenario 09, pong-controlled migrations mixed workload tests, and monitoring tests

08-09 scenario 10, pong-controlled migrations mixed workload tests, and monitoring tests

10 scenario 11, pong-controlled migrations mixed workload tests, and monitoring tests

11 scenario 12, pong-controlled migrations cpu workload tests, and monitoring tests, Sampling
Rate comparison

12 scenario 11, pong-controlled migrations mixed workload tests, and monitoring tests, G750JX
meta-host

13 Scenario 13 - baseline for null metrics, set 0 value on null metrics
14 Scenario 13 - baseline for null metrics, drop metric on null metrics

15-17 Scenario 11, pong-controlled migrations mixed workload tests, and monitoring tests after
null metric change

18 Scenario 14 - stress test, container eol test

19 Scenario 15 - Long Scenario 11 3x time, same delays

20 Scenario 16 - Long Scenario 11 3x time, 3x delays

21-22 Scenario 15-2 - Long Scenario 11 3x time, same delays

----Herdmon metric processing and sampling tests completed

23-25,27-30 Scenario 17 - first staircase cpu
26 Scenario 18 - mixed workload basedline test

26 - Scenario 18 - workload baseline test
29,28 - Scenario 17-4 - long baseline CPU test 4
22,20 - Scenario 16 - Long Scenario 11 3x time, 3x delays

30-33 Scenario 20 long baseline staircase I/O test
34 - Scenario 20-4 - long baseline I/O test 4

35-37 Scenario 21-1 long baseline test for 2x 4-core  2x 2-core  cpu test

38,39,40 vs 41-4 - Scenario 21-2 - MigrationMetrics Overhead reduction comparison
36 vs 41-4 - Scenario 21-2 - Overhead reduction comparison
42-2 vs 42-4 PCP deployment model comparison
39 vs 41-4 for preliminary and 42-4 for final comparison of optimizations

44 Scenario 20-4 Container/Host IO utilization study after updates and on a homogeneous CPU-count
VM setup, discarded
45 Scenario 20-4 Container/Host IO utilization study after updates and on a heterogeneous CPU-
count VM setup, discarded

43, 47 for stress testing for concurrency management changes
```

```
41-49 for concurrency management tests
50,51-58 for IO study
60 for cpu with stress-ng

60-4 for Tracking Thread rewrite write-up

62 Scenario 24-3 - Long baseline test for 4x 256M non-sustained memory containers - stress-ng
63 Scenario 24-4 - Long baseline test for 8x non-sustained 512M memory containers - stress-ng

64 Scenario 25-2 - Short baseline test for 4x CPU containers - stress-ng, migration tracking
tests with migration stress test
64/06 first mqtt streaming test

----Herdmon overhead optimizations completed

59,61 for ram study
s24
pong-based migrations
base metrics, no ops

65 - cpu tests, 1-2 thread count, base metrics
control host not in meta host
pong-based migrations

65/01 cfs conclusions
65/02 first forecast test?
65/03 - cpu tests, 1-2 thread count, extended metrics, init extra metrics and ops
pong-based migrations

for openstack in dcp1, dcp2 and cluster
forecasting test
resmapling test
classification test? pids vs threads

66 migration duration data to compute avg and distribution

67 first ops test
67/05 Scenario 28-2 -  long ops log test for 4 single thread cpu containers - cpu staircase
67/06 Scenario 28-2 -  long ops log test for 4 single thread cpu containers - sysbench. With
mosquitto_sub log - cpu staircase

68 - cpu tests
pong-based migrations
resampling tests

69 - mem tests, 1 thread count, base metrics, psi and ops
s30-1
pong-based migrations
resampling tests

70 - mem test
pong-based migrations

71 - cpu tests, 1 thread count, base metrics
First openstack meta-host
resampling tests

72 - cpu tests, 1 thread count, base metrics, ops
herdControl tests, init algo tests
for openstack

73 - cpu tests, 1 thread count, base and extended metrics, ops
algo 1-3 tests
for openstack
algo comparisons
```

```
74 - cpu tests, 1-2 thread count, extended metrics and ops
algo 1-3 tests
for openstack in dcp1, dcp2 and cluster
algo comparisions

75 - cpu tests, random thread count, extended metrics, extra metrics and ops
for openstack in dcp1, dcp2 and cluster

76 - stress-ng mem tests, 1 thread count, extended metrics, extra metrics, no ops
for openstack in dcp2 running on herd-1G-1 to 4

77 - mem tests, random thread count, extended metrics, extra metrics and ops
for openstack in dcp1, dcp2, cluster and dcp2 running on herd-1G-1 to 4

78 - migration stress testing after herd compute changes (network, difflayer, compression,
logging, extended metrics)
comparing sync'd criu resource vs not sync'd

----Herdmon extended metrics (PSI), extra metrics(PID), container ops and logs (from herdCompute)
completed

Folder OLTP/

TPCC 11/1 - DCP2 idle test
     11/2-11/3 Multi server oltp scenario test
     11/4-11/5 60 min Multi server oltp scenario same node
       11/5 oltp_11 base test
     11/6-7  60 min Multi server oltp scenario 6 nodes
     11/8-10 init herd control test
     11/11-12 algoTwo test
     11/13-14 algoSix test
     11/15-16 algoSix long test (oltp_11)
     11/17-19 algoSix short test (oltp_11-b)
     11/20 algoSix long test (oltp-11) after updates
     11/21 algoSeven short test (oltp-11-b) not final algoSeven
     11/22-25 algoSeven long test (oltp-11) after run of OLTP 13/12
     11/26-28 algoSeven oltp-11-d
     11/29 algoSeven oltp-11-e
     11/30 algoSeven oltp-11-f 5 WARE
     11/31 algoSeven oltp-11-f 5 WARE perfect run
     11/32 algoSeven oltp-11-g 5 WARE 15 TERM perfect run

     #disk limited (8 cores)
     11/33 oltp-11-e_2 10 WARE 10 TERM 2 clients 5 WARE each algoSeven
     11/34 oltp-11-e_2 10 WARE 10 TERM 2 clients 5 WARE base test

     #vm core count comparison
     11/35 oltp-11-h 10 WARE 10 TERM 2 clients 5 WARE staircase base test for oltp-11-e_2
8 cores each vm
     11/36 oltp-11-i 10 WARE 10 TERM 2 clients 5 WARE oltp-11-h but with                 2 cores
each vm

     #10 min test
     11/37 oltp-11-e_3 10 WARE 10 TERM 2 clients 5 WARE base test 2 cores each vm
     11/38 oltp-11-e_3 10 WARE 10 TERM 2 clients 5 WARE base test 2 cores each vm with algoSeven

---> #tpcc with oltp policy: algoSeven perfect
     11/39 oltp-11-e_4 10 WARE 10 TERM 2 clients 5 WARE base test 2 cores each vm 60min with
algoSeven
     11/40 oltp-11-e_4 10 WARE 10 TERM 2 clients 5 WARE base test 2 cores each vm 60min

     #vm core count comparison
     11/41 oltp-11-e_5 10 WARE 10 TERM 2 clients 5 WARE base test 4 cores each vm 60min
       11/42 oltp-11-e_5 10 WARE 10 TERM 2 clients 5 WARE base test 4 cores each vm 60min with
algoSeven
```

```
     11/43 oltp-11-e_5 10 WARE 10 TERM 2 clients 5 WARE base test 4 cores each vm 60min with
algoSeven second run

     #vm core count comparison
       11/44 oltp-11-e_6 10 WARE 10 TERM 2 clients 5 WARE base test 5 cores each vm 60min
     11/45 oltp-11-e_6 10 WARE 10 TERM 2 clients 5 WARE base test 5 cores each vm 60min second
run
       11/46 oltp-11-e_6 10 WARE 10 TERM 2 clients 5 WARE base test 5 cores each vm 60min with
algoSeven - failed
     11/47 oltp-11-e_6 10 WARE 10 TERM 2 clients 5 WARE base test 5 cores each vm 60min with
algoSeven

     11/48 oltp-11-e_7 10 WARE 10 TERM 2 clients 5 WARE base test 6 cores each vm 60min
       11/49 oltp-11-e_7 10 WARE 10 TERM 2 clients 5 WARE base test 6 cores each vm 60min with
algoSeven  - failed
       11/50 oltp-11-e_7 10 WARE 10 TERM 2 clients 5 WARE base test 6 cores each vm 60min with
algoSeven  - failed
     11/51 oltp-11-e_7 10 WARE 10 TERM 2 clients 5 WARE base test 6 cores each vm 60min with
algoSeven

     11/ 34,40,41,45,48 tpcc cpu count effect

     #15GB InnoDB
     14/01 10 TERM 8 cores 1-5 WARE
     14/02 10 TERM 8 cores 10 WARE
     14/03 10 TERM 8 cores 20 WARE
     14/04 10 TERM 8 cores 30 WARE
     14/05 10 TERM 8 cores 40 WARE
     14/06 10 TERM 8 cores 50 WARE
     14/07 10 TERM 8 cores 60 WARE
     14/08 10 TERM 8 cores 70 WARE fail
     14/09 10 TERM 8 cores 65 WARE <--- limit
     14/100 10 TERM 8 cores 100 WARE fail

     15/ oltp-11-e_4 10x base 2 cores
     16/ oltp-11-e_4 10x algoSeven 2 cores
     17/ oltp-11-e_4 10x base 4 cores
       17/02+ mosquitto persistence disabled
     18/ oltp-11-e_4 10x algoSeven 4 cores

OLTP   12/01 init sysbench oltp 1M-10T-T4 test (oltp_13)
12/02 init sysbench oltp 1M-10T-T100 test (oltp_13)
13/09 algoSeven (oltp_13-c) perfect results
13/10-11 oltp_15 tests, 6x sysbench, like oltp_11
13/12-13 oltp_15 tests, 6x sysbench, like oltp_11 with meta-host monitoring
13/12 oltp_15 perfect results
13/14 oltp_15-b tests, longer oltp_15, with meta-host monitoring
13/15 oltp_16, based on oltp_15 with 20 tables and 50 threads

Folder Data/

79+    herdmon baseline tests
self host, not meta, no containers 5 sec, 10 sec, 2 sec
oltp 10min 2 sec 10 sec 30 sec
oltp 60min 2 sec 10 sec 30 sec
distribution comparison, statistical moments
overhead/scaling comparison

79     herdmon baseline tests - 5s mon, 1s track, no containers, no meta-host
   94  herdmon baseline tests - 5s mon, 2s track, no containers, no meta-host
80     herdmon baseline tests - 1s mon, 1s track, no containers, no meta-host
81     herdmon baseline tests - 10s mon, 5s track, no containers, no meta-host    boxplot
82     herdmon baseline tests - 30s mon, 15s track, no containers, no meta-host   too high? redo
   93  herdmon baseline tests - 30s mon, 15s track, no containers, no meta-host, 4/8 cores, 16GB
ram, 60 sec polltimeout
83     herdmon scaling tests - 450/600 containers 1GB jvm, failed
```

```
84      herdmon scaling tests - 450/600 containers 4GB jvm - herdmon and pcp overhad box plots,
no ops
85      herdmon scaling tests - 450/600 containers 1GB jvm - herdmon and pcp overhad box plots,
cpu ops
86      herdmon scaling tests - 450/600 containers 4GB jvm - cpu ops
87      herdmon scaling tests - 600/600 containers 4GB jvm - no ops, 16GB herd-control
88      herdmon scaling tests - 600/600 containers 4GB jvm - cpu ops, 16GB herd-control
89      herdmon scaling tests - 1800 containers 8GB jvm - no ops, 16GB herd-control
90      herdmon scaling tests - 1800 containers 8GB jvm - cpu ops, 16GB herd-control
91      herdmon scaling tests - 2700/3600 containers 16GB jvm - no ops, 16GB herd-control
   95   herdmon scaling tests - 2800/3600 containers 16GB jvm - no ops, 16GB herd-control, 60 sec
polltimeout - kernel.all.pswitch vs network.interface.total.bytes?
92      herdmon scaling tests - 2700/3600 containers 16GB jvm - cpu ops, 16GB herd-control
96      herdmon scaling tests - 2800/3600 containers 16GB jvm - cpu ops, 16GB herd-control, 60
sec polltimeout
```

We developed more than 400 Jupyter notebooks for analysis, generated more than 19 thousand CSV data files, conducted more than 100 types of experiments and generated more than 1.2GB of resource utilization data.