SwarmGuardian: Towards Manager Recover Solution of Docker Swarm

Yifan Hao
yifanh5@illinois.edu

Shiming Song
ssong38@illinois.edu

Kaishen Wang
kwang40@illinois.edu

***Abstract*— Docker Swarm, as the official scheduler for Docker, is under fast construction these days. Currently, Swarm can survive from not only faults of worker nodes, but also from those of manager nodes. User can define the number of managers of Swarm, and as long as quorum of managers are alive, Swarm can act normally. However, if the Swarm loses the quorum of managers, the swarm cannot perform management tasks.[1] At the same time, adding manager nodes to a swarm makes the swarm more fault-tolerant. However, additional manager nodes harms write performance because more nodes must acknowledge proposals to update the swarm state. This implies more network round-trip traffic.[1] It is possible to have mechanisms that convert a worker node to manager node whenever a manger fails. By this, the system can keep a small number of managers, which makes Swarm more efficient while more fault-tolerant as well. We implement a system called SwarmGuardian, which monitors the liveness of managers and run tools to bring a worker up to a new manager when necessary. By keeping the number of manager nodes at minimum when Swarm is booted, and dynamically promoting worker nodes to manager nodes when necessary, we can guarantee both robustness and performance in Docker Swarm.**

I. INTRODUCTION

Containers like Docker has become more and more popular these days because of its lightweight and short start time compared to virtual machine. It is widely deployed in clusters to provide services. Docker Swarm, which is the official scheduler for Dockers, is growing more and more powerful. From non fault-tolerant, to worker node fault-tolerant, to manager node fault-tolerant, Swarm is much stronger than what it was at the first time. However, current Swarm asks user to specify the number of managers when the system is booted. Swarm can survive as long as quorum (majority) of managers are alive, but Swarm still suffer from two vulnerabilities. First, if majority of managers fail, then Swarm cannot perform management tasks. This situation is possible for those scenarios where small number of manager nodes are picked or jobs run for a very long time. Second, there exists trade-off between performance and fault-tolerance. If we want to make the system fault tolerant, it must replicate the states (worker nodes or masters) among different machines. In order to keep those replicas in a consistent state, nodes need to communicate with each other, adopt synchronization protocols to decide the order of different operations. Besides, distributed system community has a clear distinction between different levels of consistency: If the system needs to guarantee strict consistency, then every node in the system needs to be in the same state in order to serve requests from users, which implies the system cannot respond to user before nodes are synchronized; If the system adopts relaxed consistency model, even though it reduces response time, nodes might not be synchronized, and thus can provide outdated information to user. The overhead incurred by synchronization gets worse when we add more nodes to the system. Although many works have shown that relaxed consistency model is enough in certain large scale industry applications[2], there exist systems (systems in trading companies and banks, for example) requiring strong consistency model. Therefore, performance and fault tolerance are contradicts with each other by nature.

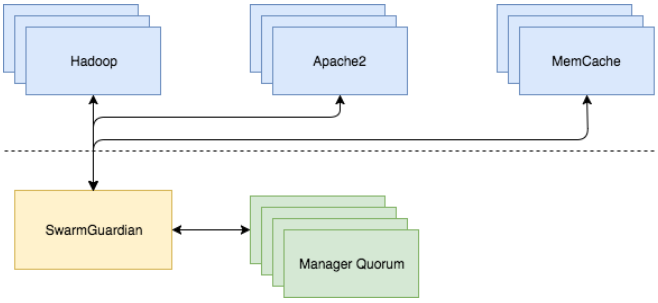

Fig. 1. A visualization of SwarmGuardian

To solve these two problems, we develop a new system SwarmGuardian and integrate it with Swarm. It will monitor all the worker nodes and manager nodes in the swarm distributed system. Whenever a manager node fails, SwarmGuardian running on other manager nodes can detect this and should be aware of other worker nodes which may be converted to new manager nodes. Then SwarmGuardian will promote one of the worker node to become the new manager node, thus keeping the number of manager nodes at constant. By using this dynamic management system, we do not need to have a large pool of managers in order to provide fault tolerance. The system could keep a small number of manager nodes at runtime, which decreases the performance overhead when deploying many manager nodes at the same time. When developing this system, we find some exciting research topics, such as the algorithms and policies to select the most appropriate nodes, tradeoff between available bandwidth and fault tolerance, etc.

One of the benefits of this system is that it utilizes the APIs provided by Docker Swarm. We implement this system as a separated tool decoupled from the Swarm kernel. Because we are not burdened with the internal implementation of Swarm, and newest Swarm provides a rich set of command line tools, the implementation is easy and fast. Besides, the design of SwarmGuardian should not introduce much

overhead and affect the performance much. We plan to perform a performance evaluation in the future.

As the result of this work, we plan to provide following contributions:

- A novel system design to dynamically promote worker node to manager node when necessary, which keep the number of manager node at minimum in order to reduce runtime latency.
- Different algorithms and policies to select the most appropriate worker node by looking at static and dynamic properties of worker nodes.
- An implementation of the designed system called SwarmGuardian, embedded with different algorithms of programmers' choice.
- An evaluation of the system tool with different selection algorithms, and demonstrate the performance gain of the best algorithm.

The rest of the report is organized as follows: Section II will briefly introduces some basic concepts behind container scheduler; Section III will compare Docker Swarm with two other existing container scheduler solutions; Section IV will describes the overall system architecture; Section V will talk about our current progress, and Section VI is about future work in this semester.

II. BACKGROUND

A. Container

Operating-system-level virtualization, also known as containerization, refers to an operating system feature in which the kernel allows the existence of multiple isolated user-space instances. Such instances, called containers[3], partitions, virtualization engines (VEs) or jails (FreeBSD jail or chroot jail), may look like real computers from the point of view of programs running in them. A computer program running on an ordinary person's computer's operating system can see all resources (connected devices, files and folders, network shares, CPU power, quantifiable hardware capabilities) of that computer. However, programs running inside a container can only see the container's contents and devices assigned to the container.

On Unix-like operating systems, this feature can be seen as an advanced implementation of the standard chroot mechanism, which changes the apparent root folder for the current running process and its children. In addition to isolation mechanisms, the kernel often provides resource-management features to limit the impact of one container's activities on other containers.[4]

In general, a container image is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it, including code, runtime, system tools, system libraries, settings. Available for both Linux and Windows based apps, containerized software will always run the same, regardless of the environment. Containers isolate software from its surroundings, for example differences between development and staging environments and help reduce conflicts between teams running different software on the same infrastructure.[5]

B. Docker

Docker is a software technology providing containers, promoted by the company Docker, Inc.[6] Docker provides an additional layer of abstraction and automation of operating-system-level virtualization on Windows and Linux.[7] Docker uses the resource isolation features of the Linux kernel such as cgroups and kernel namespaces, and a union-capable file system such as OverlayFS and others[8] to allow independent "containers" to run within a single Linux instance, avoiding the overhead of starting and maintaining virtual machines (VMs).[9]

The Linux kernel's support for namespaces mostly[10] isolates an application's view of the operating environment, including process trees, network, user IDs and mounted file systems, while the kernel's cgroups provide resource limiting, including the CPU, memory, block I/O, and network. Since version 0.9, Docker includes the libcontainer library as its own way to directly use virtualization facilities provided by the Linux kernel, in addition to using abstracted virtualization interfaces via libvirt, LXC (Linux Containers) and systemd-nspawn.[11] [12] [13] [14]

C. Swarm

Docker Swarm is a clustering and scheduling tool for Docker containers. With Swarm, IT administrators and developers can establish and manage a cluster of Docker nodes as a single virtual system.

Swarm mode also exists natively for Docker Engine, the layer between the OS and container images. Swarm mode integrates the orchestration capabilities of Docker Swarm into Docker Engine 1.12 and newer releases.

Clustering is an important feature for container technology, because it creates a cooperative group of systems that can provide redundancy, enabling Docker Swarm fail over if one or more nodes experience an outage. A Docker Swarm cluster also provides administrators and developers with the ability to add or subtract container iterations as computing demands change.

An IT administrator controls Swarm through a swarm manager, which orchestrates and schedules containers. The swarm manager allows a user to create a primary manager instance and multiple replica instances in case the primary instance fails. In Docker Engine's swarm mode, the user can deploy manager and worker nodes at runtime.

Docker Swarm uses the standard Docker application programming interface to interface with other tools, such as Docker Machine. [15]

If the swarm loses the quorum of managers, the swarm cannot perform management tasks. If your swarm has multiple managers, always have more than two. In order to maintain quorum, a majority of managers must be available. An odd number of managers is recommended, because the next even number does not make the quorum easier to keep. For instance, whether you have 3 or 4 managers, you can still only lose 1 manager and maintain the quorum. If you have 5 or 6 managers, you can still only lose two.

Even if a swarm loses the quorum of managers, swarm tasks on existing worker nodes continue to run. However, swarm nodes cannot be added, updated, or removed, and new or existing tasks cannot be started, stopped, moved, or updated. [16]

III. RELATED WORKS

There are other container schedulers developed by different organizations. The most popular ones are Kubernetes from Google, and Mesos from Apache Software Foundation. This section gives a brief introduction to these two schedulers, and compare them with Docker swarm.

A. Mesos

Apache Mesos is more than a container scheduler. It is a distributed system kernel built with the same philosophy as Linux kernel, but at different layer of abstraction. The Mesos kernel runs on every machine and provides applications (e.g., Hadoop, Spark, Kafka, Elasticsearch) with APIs for resource management and scheduling across entire datacenter and cloud environments.[17]

The purpose of Mesos is to build a scalable and efficient system that supports a wide array of both current and future frameworks. This is also the main issue: frameworks like Hadoop and MPI are developed independently thus it is not possible to do fine-grained sharing across frameworks.[18]

Mesos leverages features of the modern kernel"cgroups" in Linux, "zones" in Solaristo provide isolation for CPU, memory, I/O, file system, rack locality, etc. The big idea is to make a large collection of heterogeneous resources. Mesos introduces a distributed two-level scheduling mechanism called resource offers. Mesos decides how many resources to offer each framework, while frameworks decide which resources to accept and which computations to run on them. It is a thin resource sharing layer that enables fine-grained sharing across diverse cluster computing frameworks, by giving frameworks a common interface for accessing cluster resources.The idea is to deploy multiple distributed systems to a shared pool of nodes in order to increase resource utilization. A lot of modern workloads and frameworks can run on Mesos, including Hadoop, Memecached, Ruby on Rails, Storm, JBoss Data Grid, MPI, Spark and Node.js, as well as various web servers, databases and application servers.[19]

Mesos uses a fail-fast approach to error handling: if a serious error occurs, Mesos will typically exit rather than trying to continue running in a possibly erroneous state. For example, when Mesos is configured for high availability, the leading master will abort itself when it discovers it has been partitioned away from the Zookeeper quorum. This is a safety precaution to ensure the previous leader doesnt continue communicating in an unsafe state.

To ensure that such failures are handled appropriately, production deployments of Mesos typically use a process supervisor (such as systemd or supervisord) to detect when Mesos processes exit. The supervisor can be configured to restart the failed process automatically and/or to notify the cluster operator to investigate the situation. The focus of Mesos is on scalability and abstraction rather than fault tolerant. An operator needs to manually configure the system when failure occurs.

B. Kubernetes

Kubernetes is an orchestration system for Docker containers using the concepts of labels and pods to group containers into logical units. Pods are the main difference between Kubernetes and the two other container schedulers. They are collections of co-located containers forming a service deployed and scheduled together. This approach simplifies the management of the cluster comparing to an affinity based co-scheduling of containers (like Swarm and Mesos).[18]

Kubernetes is more complex than Swarm, but it offers more features to users. Kubernetes extensively use YAML files to define all services in the cluster, including deploying pods, services and replica controllers. The YAML configuration is unique to Kubernetes. It suits a medium to large clusters, and it is very well suited for complex applications with many containers inside pod. Besides, according to official website of Kubernetes, it is also capable of automatic binpacking, automatic rollouts and rollbacks, secret configuration management. One of the related features provided by Kubernetes is self healing. Kubernetes is capable of restarting containers that fail, replacing and rescheduling containers when nodes die, killing containers that don't respond to clients through user-defined health check. However, to the best of our knowledge, the self healing feature in Kubernetes does not handle the scenario where some of the manager nodes fail.[20]

C. Summary

The reason for us to improve Docker Swarm rather than the other two solutions is largely for simplicity. Docker is one of the most popular containers used in the community, and is open sourced. Docker and Swarm are developed by the same organization, and they share the same APIs, which means they are perfectly compatible with each other. Docker Swarm also has the fame in simple usage. Mesos and Kubernetes work well when applied to large scale clusters, but they suffer from complicated setup. Because the manager failure detection and promotion discussed in this report is only for proof of concept, we choose to develop this feature on Docker Swarm to accelerate the research and experiment process.

IV. APPROACH AND SYSTEM DESIGN

To achieve the goal of keep a constant number of manager nodes automatically, we need to complete several subtasks.

First, we need to develop tools to make manager nodes be aware of any failure of other manager nodes. Then the system needs to decide which manager node will run the promotion decisions. Consistency is necessary, if more than one managers make decisions and they decide to promote different worker nodes, the situation could be messy. Decision about which worker node should be promoted also worths

much consideration. In general, as the Swarm documentation suggests, for optimal fault-tolerance, distribute manager nodes across a minimum of 3 availability-zones to support failures of an entire set of machines or common maintenance scenarios.[21] At last, we also want to handle different extreme circumstances. For instance, what should we do if there's no available worker nodes available? what should we do if there's few nodes available?

We will look at each step at each section carefully, and discuss about our approach to these problems.

A. Failure Detection

The first step is to detect any failure on any manager nodes. Current Docker Swarm provides interfaces which can give information about the reachability of each manager node[22]. If any manager nodes fails, it becomes unreachable from other managers. We develop a tool based on these interfaces. The tool will run those commands periodically, and analyze the results of those commands to check the liveness of other manager nodes.

There's a tradeoff in the design of frequencies of running those commands in our tool. If we make the tool running the check very often, such additional work may introduce some overhead to the machine, thus affect the performance of other tasks. However, if we make the tool running the check once for a very long time, it increase the possibility of losing quorum of manager nodes and failure detection time during that period. We will set up a frequency based on our experimental experience, and also enable user to define their own.

B. Manager Node Choice

We also need to keep consistent about which manager node will do the promotion task. We can make it simple and leave this job to the only leader in the system, which is also called the primary manager. If a leader fails, the RAFT consensus in the docker swarm will make an election and decide a new leader[23]. If we have different strategies dealing with normal manager node failure and leader failure, then we may encounter the situation where we have made a decision on a non-leader manager, and a new leader is promoted and consider itself to be the one to do the promotion tasks. To avoid the conflict of manager decision caused by this situation, we will just wait until a new leader is decided. So it is always the leader who makes the promotion decision.

C. Worker Node Choice

As for worker node choice, it is the most complicated phase in our recovery mechanism. In general, we want to make the manager nodes in as many different partition zones as possible, which can decrease the possibility that single rack/ data center failure will cause multiple manager nodes failure.

A simple way to keep track of which data center each node belongs to is to include the name of data center in the node ID during the creation of each node[24]. The leader can make a decision to promote a worker node which doesnt belong to any data center that already hold any existing manager nodes.

In addition, we also want to forcibly demote the unreachable manager node before we promote the worker node, in case the dead manager node is recovered and keeps his manager role in the system. In such case, there will be more manager nodes than the user specified, which is undesired.

D. Special Cases

Currently, the mechanism can handle most normal manager node failure situations. However, there are special cases we need to pay attention to. Well discuss several special cases and provide our solution for solving these situations:

1) No worker node: Under this certain circumstance, the system contains manager node only . At this time, if any manager node crashes, no promotion will happen this time since there is no pure worker node in the system. The new node can still join the system. Our mechanism will be effective when we have a certain number of nodes. This situation will be discussed in other part.

2) Very few worker node: Under this circumstance, the trade-off between promoting a worker node to the manager and decreasing the amount of manager node should be considered. Several different strategies will be discussed below:

a. Fault-tolerance guaranteed

By looking at the design of the failure tolerance mechanism of the swarm docker, the system containing N manager nodes can tolerate the loss of at most (N-1)/2 managers. If the amount of managers is less than the (N-1)/2, the recovery has to been done by the user. For the purpose of guaranteeing the failure-tolerance feature, we can keep the amount of manager nodes we have relatively larger than the least amount of manager nodes which are required to guarantee failure tolerance. Currently, we choose 3(N-(N-1)/2)/4 + (N-1)/2 as the threshold numbers according to the least number which keeps the system running. Later, we will test the performance based on selected threshold number to check whether it will offer the best performance or not. If the amount of worker nodes is less than and equal to the threshold number we selected, we will have two different situations to consider. If the manager node is the manager node only, we need to at least have one worker node. Others will be promoted to the manager node. If the manager node works as both the manager node and worker node, all worker nodes will be promoted as the manager nodes. If the amount of worker nodes is larger than the selected thresholds, the mechanism we are discussing above will promote the certain numbers of worker nodes to the manager nodes.

b. Performance guaranteed

By considering the performance of the system, the only factor we need to consider is that do we really need more manager nodes in the system. Compared with the pure worker node, the manager node which played two roles will have the heavy load and execute less tasks than the pure worker node. Thus, the performance will be different. Our strategy depends on what role manager node played in the

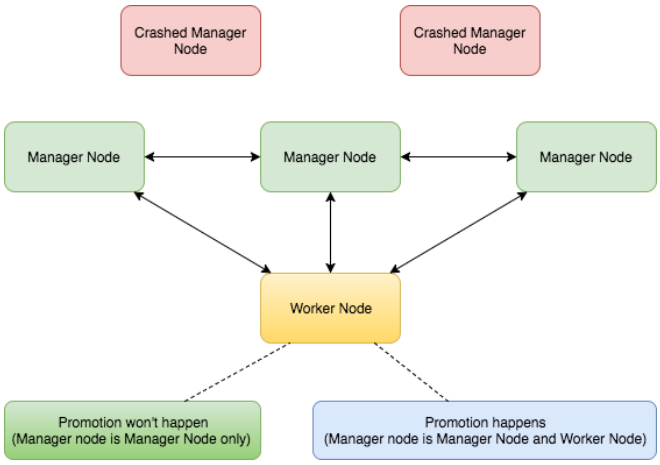

Fig. 2. A visualization of Fault-tolerance Guaranteed(Very few nodes) case

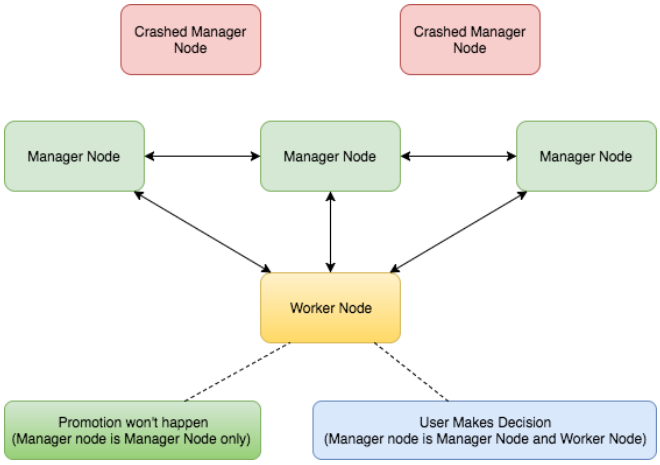

Fig. 3. A visualization of Performance Guaranteed(Very few nodes) case

system. If the manager node is the manager node only, we need to have enough work nodes in the system to execute the task. Thus, if the amount of manager nodes is larger than the worker node, our mechanism wont allow the manager node to promote the worker node to be the manager node. If not, our mechanism will work as usual automatically. If the manager node works as the manager node and worker node, our mechanism wont make any decision if the amount of manager nodes is larger than the worker node. It will send the email notification to let the user know the situation, provide enough information and make the user make their decision.

In our mechanism, the user has to decide if he/she prefer fault-tolerant feature or performance feature by himself/herself.

V. IMPLEMENTATION

A. Overview

We implemented SwarmGuardian in Python. We used subprocess call to retrieve necessary information about all nodes in Swarm, scanned all nodes information to

```
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS
bknt7befvkfm9wesfw8cmiidy     dc0-0               Ready               Active              Leader
ut6s0nvb9v54h1dscf8rklrct     dc0-1               Down                Active
wmqqwfgxh5mcapc7ziwbtrpio     dc0-1               Ready               Active              Reachable
uapxpaln27sumf9ebawuzd2rd     dc0-1               Down                Active
z775zl8bbaiahcqigfe6lwgq2     dc0-2               Down                Active
mljreig4psny4ahypqbmfkmsv     dc0-2               Down                Active
d0iv5y8th8nkaxhfnr8jeifmz     dc0-2               Down                Active
r06oukkky8al8pz58y4sclxle *   dc1-0               Ready               Active              Reachable
```

Fig. 4. A visualization of Fault-tolerance Guaranteed(Very few nodes) case

understand the condition of Swarm, and made decisions according to the design we described earlier. We completed the full implementation in several hundreds lines of code. Our implementation can be found at https://github.com/swarmPaxos/SwarmGuardian.

B. Configuration File

The file provides user with a way to personalize SwarmGuardian according to different user purposes. The configuration file will be in json format and has four fields. An example of configuration file may looks like this:

```
{
"SLEEP_SECS": 5,
"DESIRED_MANAGER": 3,
"POLICY": 1,
"EMAIL": "user@example.com",
"CACHE": 1
}
```

SLEEP_SECS is the value that how frequent the user wants to run SwarmGuardian. The time unit is second. By default, the value is 5.

DESIRED_MANAGER is the value that how many manager nodes the user wants to keep in the Swarm Docker. By default, the value is 3.

POLICY is the value representing the different policy SwarmGuardian will follow if the extreme case happened. 0 is referred as fault-tolerance setting and 1 is referred as performance setting. By default, the value is 0.

EMAIL is the email address that the user wants to get the notification from the SwarmGuardian.

CACHE indicates whether caching is enabled in SwarmGuardian, which will be covered in implementation section. 0 means caching is disabled, otherwise enabled.

C. Node Status List

Node Status List is the list containing the current status of all nodes in the system, which is a really useful way to verify whether our functionality works. There are five fields which is listed and explained below except AVAILABILITY since it isnt used in our mechanism:

ID is the value that used to distinguish between different machines. This value is randomly assigned by the SwarmDocker.

HOSTNAME is the name which user defines. By using SwarmGuardian, user is required to name their home using format dcX-Y. X represents the id of the data center. Y represents the machine in the system. For example, by looking at the first row in the figure 5, dc0-0 represents data center 0 machine 0. This field is important to our promotion algorithm.

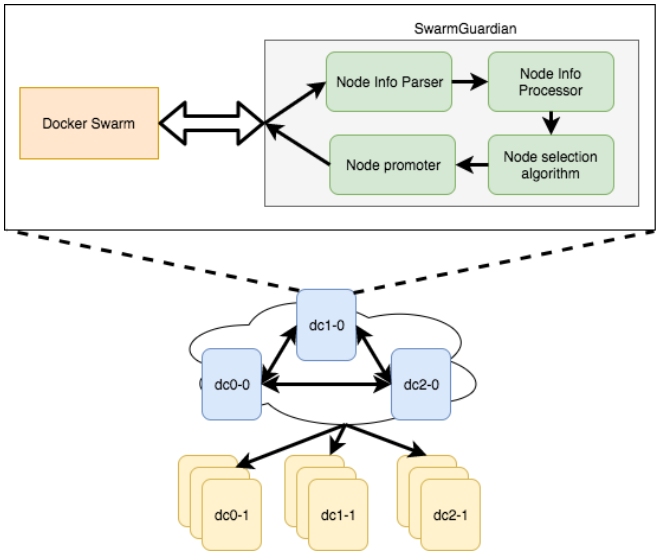

Fig. 5. A visualization of the system architecture. All the blocks at the bottom are servers with name indicating their clusters. The green blocks on the top are components in SwarmGuardian, and it interfaces with Docker Swarm

STATUS is the status of the machine. Ready represents that the machine is still alive. Down represents that machine is not alive.

MANAGER STATUS is the value showing if the node is manager node and what role it plays in the manager quorum. There are three values for this field. Leader means the node is the leader. Reachable means the node is one of manager nodes, but not the leader. Empty means that the node is not the manager. This field is important for the evaluation work of our mechanism.

D. Architecture

As indicated in Figure 5, SwarmGuardian will request node list status information from Docker Swarm with period defined by user. SwarmGuardian will feed those information to node information parser in the pipeline. The extracted information will then be processed by SwarmGuardian. As SwarmGuardian processes the information, it will recognize the number of the alive managers and the liveness of all the worker nodes. Once SwarmGuardian is done with the processing work, and it finds out that currently alive managers are less than the desired number of managers, it will start the node selection engine to select the best worker node. The node promoter will then based on the result to invoke Docker Swarm API and promote the right worker node.

One interesting problem arose when we implemented our system: what should we do with the manager node that just appeared to be dead? Various problems could happen in the manager node, and there is no way we could tell if the manager appeared to be dead because of the network partition or crashes within the manager. If SwarmGuardian does not do anything to the manager node that just went down, the manager node might come back, which will cause the number of currently available manager nodes more than desired number of manager nodes. To solve this problem, SwarmGuardian can simply demote that manager node. This can solve the issue because it will become a dead worker node, and it will not take a position in manager's quorum. Besides, even if that manager node comes back, it will act as a normal worker node, which will then be promoted as a manager node again if there is no other appropriate candidates.

E. Optimization

One major problem of the promotion algorithm is that each time SwarmGuardian is invoked, it needs to scan through the node status list to check what worker nodes are available, such that when a manager node fails, leader can promote available worker node to be future manager. The complexity to scan through the list is O(n), where n is the number of nodes in the list. The runtime overhead of the scanning procedure will severely degrade the performance of the tool as the number of servers in cluster becomes larger.

To overcome this issue, we made following observation: the average time for a server to fail usually takes years. Therefore, once the tool finds that a server is alive in certain period, it can assume the server will continue live for some period due to temporal locality. Based on this observation, we implement a caching mechanism in SwarmGuardian. SwarmGuardian internally holds a caching queue. If the queue is empty, SwarmGuardian will go ahead and invoke the promoting procedure. The promoting procedure will add worker nodes into caching queue. When SwarmGuardian finds another failure, it will promote worker node from caching queue without invoking promotion procedure. It is possible that the cached worker node is dead when we try to promote it. In this case, the promotion will fail, and SwarmGuardian will find the next cached worker node and promote it. If no available cached worker nodes exist in caching queue, the promotion procedure will be invoked again.

VI. Evaluation

We conducted an experiment with 7 EC2 instances from AWS (in Figure 7).

First, we have verified the correctness over many different circumstances.

We intend to test with different number of manager nodes and worker nodes, and test with different policies (fault-tolerant policy and performance guaranteed policy). We have tested the following scenarios:

Then we use "htop" command to monitor the resources used by our tool. The tool has so little overhead that the resources usage is dominated by the minimal usage of running a process. In all cases we have tested, with change in number of nodes and sleep duration, for a single tool running on a EC2 instance, the percentage of CPU usage is 0.0%, the percentage of memory usage is 0.8%, the actual physical memory cost is 3936 KB.

Limited by the resources we have, we cannot measure the accurate overhead of running out tool. However, current

Total Nodes	Manager Nodes Required	Leader Fail / Normal Manager Fail	Performance Guaranteed / Fault-tolerance Guaranteed	Success or Not
6	4	Leader Fail	Doesn't matter	Yes
6	4	Normal Manager Fail	Doesn't matter	Yes
5	3	Leader Fail	Doesn't matter	Yes
5	3	Normal Manager Fail	Doesn't matter	Yes
4	3	Leader Fail	Performance Guaranteed	Yes
4	3	Leader Fail	Fault-tolerance Guaranteed	Yes
4	3	Normal Manager Fail	Performance Guaranteed	Yes
4	3	Normal Manager Fail	Fault-tolerance Guaranteed	Yes

Fig. 6. Verification Scenarios. Each row is a scenario with different number of managers and workers.

measurement is enough to show that running our tool has very little overhead and will not affect the system much even in a larger scale.

Fig. 7. Ubuntu Server 16.04 LTS(HVM), SSD Volume Type-ami-aa2ea6d0, Memory: 1 GB

VII. FUTURE WORK

We have already shown the correctness and plausibility of SwarmGuardian. In the future, there are several possibilities which SwarmGuardian can be exploited in the future:

1. The functionality of SwarmGuardian can be incorporated into Docker Swarm system, which will further minimize the overhead of it. SwarmGuardian can be considered to be implemented as the API in the future and apply to more container mechanism which doesnt have the automatic manager node election mechanism.

2. The functionalities can be modified to have more flexibility.

3. SwarmGuardian should be tested with larger scale setting like managing hundreds of nodes in Swarm. Currently we only have access to 7 instances of EC2, and we did not test our system thoroughly in large scale clusters. The correctness and efficiency should be ensured.

4. Currently, SwarmGuardian cannot remove node from the node status list after the node crashed/left the system. In our design, these zombie nodes increase the overhead and the actually running time of the algorithm. Therefore, in the future, SwarmGuardian is supposed to have the ability to remove the crash/leave node from the node status list. Meanwhile, SwarmGuardian should ensure that rejoin node isn't removed from the list.

5. Current method to determine the location and data center that each node belongs to is combining these information into their names. However, there can be a more general approach to this. One possible idea is that for every node, when joining Docker Swarm, it should provide its MAC address, node ID, location and data center information to others. Then the promotion can be determined based on such information. Even if a node is dead and rejoin Swarm later, new information will be provided. Furthermore, living managers can build cache about such information by memorizing the connections between unique MAC addresses and their physical locations.

VIII. CONCLUSION

In this project, we designed and built a system called SwarmGuardian. It is decoupled from the core of Docker Swarm system to dynamically adjust the pool of manager nodes. We also implemented a caching technique to accelerate the promotion procedure, and make the system practical in large scale clusters. By giving this dynamicity to Swarm, we can achieve both robustness and performance at the same time.

REFERENCES

[1] "Administer and maintain a swarm of Docker Engines", Docker Documentation, 2017. [Online]. Available: https://docs.docker.com/engine/swarm/admin_guide/#operate-manager-nodes-in-a-swarm. [Accessed: 21- Oct- 2017].

[2] Giuseppe DeCandia , Deniz Hastorun , Madan Jampani , Gunavardhan Kakulapati , Avinash Lakshman , Alex Pilchin , Swaminathan Sivasubramanian , Peter Vosshall , Werner Vogels, Dynamo: amazon's highly available key-value store, Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, October 14-17, 2007, Stevenson, Washington, USA

[3] Hogg, Scott (2014-05-26). "Software Containers: Used More Frequently than Most Realize". Network World. Network World, Inc. Retrieved 2015-07-09. There are many other OS-level virtualization systems such as: Linux OpenVZ, Linux-VServer, FreeBSD Jails, AIX Workload Partitions (WPARs), HP-UX Containers (SRP), Solaris Containers, among others.[Accessed: 21- Oct- 2017].

[4] [6]"Operating-system-level virtualization", En.wikipedia.org, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Operating-system-level_virtualization. [Accessed: 21- Oct- 2017].

[5] "What is Docker?", Docker, 2017. [Online]. Available: https://www.docker.com/what-docker. [Accessed: 21- Oct- 2017].

[6] Vivek Ratan (February 8, 2017). "Docker: A Favourite in the DevOps World". Open Source Forum. Retrieved June 14, 2017.

[7] O'Gara, Maureen (26 July 2013). "Ben Golub, Who Sold Gluster to Red Hat, Now Running dotCloud". SYS-CON Media. [Accessed: 21- Oct- 2017].

[8] "Select a storage driver documentation". Docker documentation. Archived from the original on 2016-12-06. [Accessed: 21- Oct- 2017].

[9] "Docker Documentation: Kernel Requirements". docker.readthedocs.org. 2014-01-04. Archived from the original on 2014-08-21. [Accessed: 21- Oct- 2017].

[10] Dan Walsh. "Yet Another Reason Containers Don't Contain: Kernel Keyrings". projectatomic.io. [Accessed: 21- Oct- 2017].

[11] Steven J. Vaughan-Nichols (2014-06-11). "Docker libcontainer unifies Linux container powers". ZDNet. [Accessed: 21- Oct- 2017].

[12] "libcontainer - reference implementation for containers". github.com. [Accessed: 21- Oct- 2017].

[13] "Docker 0.9: Introducing execution drivers and libcontainer". docker.com. 2014-03-10. [Accessed: 21- Oct- 2017].

[14] [8]"Docker (software)", En.wikipedia.org, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Docker_(software). [Accessed: 21- Oct- 2017].

[15] "What is Docker Swarm? - Definition from WhatIs.com", SearchITOperations, 2017. [Online]. Available: http://searchitoperations.techtarget.com/definition/Docker-Swarm. [Accessed: 21- Oct- 2017].

[16] ”Administer and maintain a swarm of Docker Engines”, Docker Documentation, 2017. [Online]. Available: https://docs.docker.com/engine/swarm/admin_guide/#maintain-the-quorum-of-managers. [Accessed: 21- Oct- 2017].
[17] ”Apache Mesos”, Apache Mesos, 2017. [Online]. Available: https://mesos.apache.org/. [Accessed: 21- Oct- 2017].
[18] ”Comparison of Container Schedulers Armand Grillet Medium”, Medium, 2017. [Online]. Available: https://medium.com/@ArmandGrillet/comparison-of-container-schedulers-c427f4f7421. [Accessed: 21- Oct- 2017].
[19] ”Open source datacenter computing with Apache Mesos”, Opensource.com, 2017. [Online]. Available: https://opensource.com/business/14/9/open-source-datacenter-computing-apache-mesos. [Accessed: 21- Oct- 2017].
[20] ”Kubernetes”, Kubernetes, 2017. [Online]. Available: https://kubernetes.io/. [Accessed: 21- Oct- 2017].
[21] Administer and maintain a swarm of Docker Engines. [online] Available at: https://docs.docker.com/engine/swarm/admin_guide/#distribute-manager-nodes [Accessed 21 Oct. 2017].
[22] ”Administer and maintain a swarm of Docker Engines”, Docker Documentation, 2017. [Online]. Available: https://docs.docker.com/engine/swarm/admin_guide/#monitor-swarm-health. [Accessed: 21- Oct- 2017].
[23] ”Raft consensus in swarm mode”, Docker Documentation, 2017. [Online]. Available: https://docs.docker.com/engine/swarm/raft/. [Accessed: 21- Oct- 2017].
[24] [4]2017. [Online]. Available: (https://docs.docker.com/engine/swarm/services/#create-services-using-templates). [Accessed: 21- Oct- 2017].