Live Container Migration

Abhishek Gupta, Rithvik Chandan, Tanmay Gujar
UC San Diego

1 Problem Definition

A large number of containerized applications are managed using tools like Podman that make cluster management easier through their abstractions over containers. They also support a feature that benefits containers running long jobs: The ability to move containers to other hosts with minimal runtime (Live Migration) so that multiple containers can be packed on one host for better resource utilization, costs, and performance. At present, this live migration, although user-friendly, is required to be manually triggered. We aim to design and implement an automated mechanism for scheduling containers across a pool of virtual machines. It uses CPU utilization of containers to migrate containers from a VM with higher CPU utilization to a lower one using a greedy scheduling algorithm.

Our code can be found at: https://github.com/tgujar/live-migrate

2 Design and Implementation

The architecture of the system is shown in Fig. 1. Each VM has a cron job running that regularly runs the **podman stats** command to receive the CPU utilization of each container. Using this information, the job computes a difference between the previous and current CPU statistics. It then sends this **Statistic Diff** to the controller. The reason behind computing diffs is to filter out statistics that are not crucial. For example, a change in CPU utilization by x% can be ignored. This has not been implemented yet but is easy to incorporate into the current architecture. The diffs are sent to a RabbitMQ queue to ensure theres an inorder consumption of diffs by the controller when receiving diffs from multiple VMs. The controller then updates the centrally managed statistics that contains statistics for each VM.

Each VM also has an APIServer running that performs two tasks. First, it sends a heartbeat to the controller ensuring that the controller has information about only those VMs that are active and subscribed to the system. The controller runs a maintainer that removes a particular VM from its list if it hasnt received a heartbeat in the last 2 minutes. Apart from this, the API server also does the job of container migration. At scheduled intervals, the controller runs a schedule functionality that decides migration of containers using a greedy policy. It then triggers a checkpoint and restore API call to the APIServer running on each VM that performs the checkpoint-

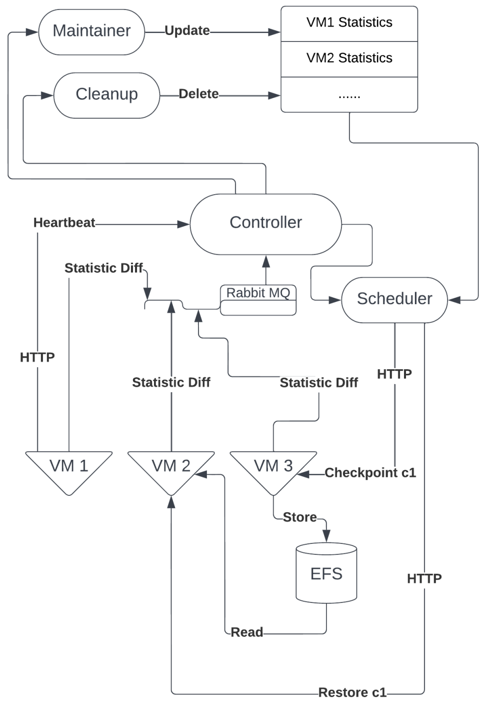


Figure 1: System Architecture

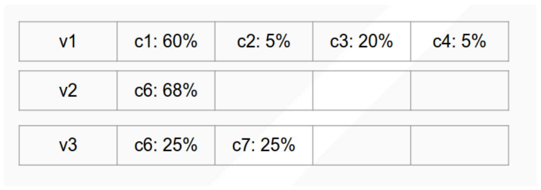
ing and restoring functionality for container migration. The APIServer also exposes functionality to create and run a container on a VM given the image name using HTTP calls.

Checkpoint files are stored on AWS EFS and are globally available to any container in the cluster. This removes the need for coordination between servers. The controller only needs to coordinate checkpointing on a container and restoring on another and doesn't concern itself with how checkpoint and restore is actually performed. This improves the flexibility of the system since the migration, checkpoint, restore logic can be changed without modifying the controller.

The system supports rootful Podman[1] containers based on `runc`[3] container runtime. `runc` runtime internally uses CRIU to support container checkpointing and restore. Podman then exposes a high level API and adds support for container resource usage metrics, and in general provides a container management solution. The servers on which the containers are running also need to use Linux kernel version 6.0.0 or higher, since a lot of the syscall functionality required by CRIU is not implemented in earlier versions of the kernel. The system also requires `runc`v1.0.0 or higher and Podman v4.2 release or higher.

2.1 Scheduling Policy

The scheduler works on a greedy approach to scheduling. We maintain an internal mapping in the architecture, called VMStats, which maintains the mapping for each VM to its constituent containers. Each of these container fields is a key-value pair which has the container ID as the key and the percentage of CPU usage of the container as the value. The mappings look like the example in the figure below:



v1	c1: 60%	c2: 5%	c3: 20%	c4: 5%
v2	c6: 68%			
v3	c6: 25%	c7: 25%		

We assume that each of the VMs have the same amount of RAM capacity in their CPU. Thus, we simply associate the containers as using a percentage of the total capacity of each VM, where each VM has the same capacity. This assumption greatly simplifies our model, since we only have to deal with percentages rather than absolute values. However, this was done only keeping the time-frame in mind and because we had to focus on other, more pressing issues (which we discuss in other sections). We believe that our scheduler can be easily enhanced to support VMs having different CPU sizes. We are easily able to take the percentages of each container and calculate the RAM they will use by multiplying these percentages with the total memory of the VM. Then, if we keep the total size of each VM in mind, we can apply our greedy algorithm with absolute memory values against the absolute memory of each VM. Thus, we feel extending our algorithm to clusters with non-homogeneous VMs in terms of memory is trivial.

Now, we move on to the 2 scheduling algorithms we use, once to initially schedule the containers on the VMs, and then the periodic checking for optimial placement and scheduling based on the periodic metric gathering of VM statistics by our controller.

2.1.1 Initial Scheduling Policy

Initially, the VMs have random containers scheduled on them that may not be the most optimal. When the controller starts, it first calls the initial scheduling algorithm. Here, a complete overhaul of the container placement is done, so that there is optimal load balancing in each VM. While this can be an expensive step at first, it ensures that the later load balancing required is minimized and thus the system runs in optimal state with minimal downtime later.

When the initial VM to container (key: container ID and value: Container CPU usage) for each VM is taken by the scheduler, all the containers from all the VMs are first listed and sorted in decreasing order of CPU usage. The VMs are presumed to be empty, and then the containers are scheduled on each VM based on decreasing order of the remaining CPU space on each VM. Thus, for example, for 2 VMs v1 and v2, a container taking 40% of CPU would first be scheduled on v1, and then 2 containers taking 20% and 25% respectively. This continues till all the containers have been scheduled on a VM. Finally, this updated map is compared to the original map and a container migration mapping, which is a map with key value pair as (containerID: (source VM ID, target VM ID), is generated. This mapping is sent to the container migration workload to do the actual migration. This initial migration step ensures that the load is equitably distributed across all VMs and subsequent scheduling needs are minimized as the system is in the most optimal state.

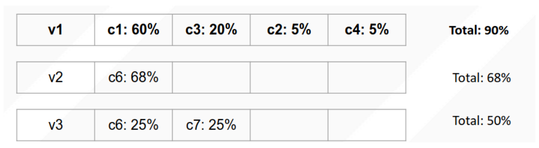
While this system works fine in the general case, we noticed issues sometimes when calling the initial scheduling algorithm right after starting controller, which were caused due to Podman taking time in scheduling our map sometimes. At the end of the term, we didn't have time to debug these issues. Thus, we have detached this Initial Scheduling code from our main code and do not call this function, focusing on making sure our application doesn't crash instead of focusing on a slight performance gain.

2.1.2 Generalized Scheduling in Operation

In the general case, we use a greedy algorithm to make scheduling decisions for determining if any container needs to be migrated. The scheduler runs as a daemon process in a separate go routine, periodically every 1 minute. This time period was chosen through experimentation to ensure that the network was not overwhelmed, the scheduler had time to make decisions and that the load balancing could occur on time to avoid a cascading failure. As a reminder, the main aim of our load balancing application is to prevent cascading failures. These would happen when, say one VM is filled to 98% capacity, when one of its containers loads increases, it would cause the whole VM to fail. This would mean the containers would have to be scheduled on other VMs, which might lead to their CPU capacity being overwhelmed, which would lead to a cascading failure of VMs. Our scheduler prevents this cascading failure scenario.

We want our scheduler, whenever it runs periodically, to do the optimal load balancing of containers over the VMs to avoid a failure scenario. However, if we do an optimal load balancing every time, it may lead to unnecessary scheduling overhead. For example, if each VM has less than 50% of its CPU being used, the probability of a container increasing its load and failing the VM is very low. In this case, an optimal scheduling would just add an overhead without giving any benefits. To mitigate this, we provide a target percentage to the scheduling algorithm. The container migration is done for only those source VMs whose CPU usage exceeds this target, thus bringing them out of the danger zone.

Our greedy strategy begins by first sorting the containers in each VM in decreasing order according to their CPU usage (per VM, not all the containers as in the case of the Initial Scheduling). We then identify the VMs that have their CPU usage above target. Taking the example we have from before, in the figure, we see that for a target of 71%, v1 needs to be optimized.

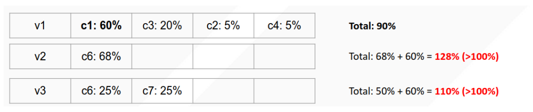


v1	**c1: 60%**	**c3: 20%**	**c2: 5%**	**c4: 5%**	**Total: 90%**
v2	c6: 68%				Total: 68%
v3	c6: 25%	c7: 25%			Total: 50%

We start from the most CPU intensive container, and try to schedule it on other VMs. To ensure optimality, we have 2 conditions:

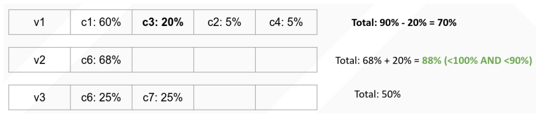
- Scheduling the container on another VM should not exceed that VM's CPU capacity (to make migration possible)
- The sum of this container's workload and the target VM's current workload should not exceed the source VM's current workload (with the container). For example, if the source VM is at 90% with this container, and moving this container to the target VM makes its workload 92%, we do not consider that a valid scheduling, and keep searching. This ensures optimality.

Applying these theories to our example, we see that for the VM v1, Scheduling container c1 on any of the other VMs violates our condition 1.



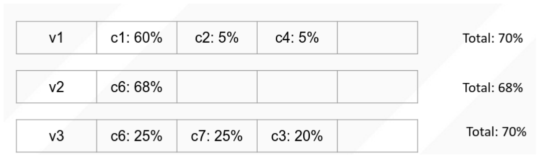
v1	**c1: 60%**	c3: 20%	c2: 5%	c4: 5%	**Total: 90%**
v2	c6: 68%				Total: 68% + 60% = **128% (>100%)**
v3	c6: 25%	c7: 25%			Total: 50% + 60% = **110% (>100%)**

We see that then scheduling container c3 on VM v2 satisfies all our above conditions. This optimizes our placement for v1, however, now v2 becomes a problematic VM.



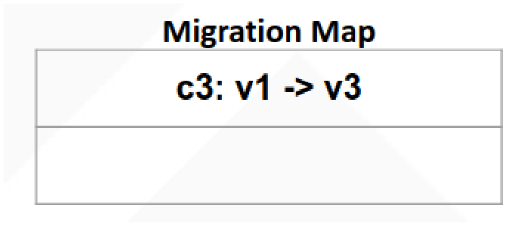
v1	c1: 60%	**c3: 20%**	c2: 5%	c4: 5%	**Total: 90% - 20% = 70%**
v2	c6: 68%				Total: 68% + 20% = **88% (<100% AND <90%)**
v3	c6: 25%	c7: 25%			Total: 50%

We then repeat the same process for v2's containers. Finally, we find the following optimal placement.



v1	c1: 60%	c2: 5%	c4: 5%		Total: 70%
v2	c6: 68%				Total: 68%
v3	c6: 25%	c7: 25%	c3: 20%		Total: 70%

We then get the final migration map of the form (containerID: (source VM ID, target VM ID), which for our example, looks as follows:



Migration Map
c3: v1 -> v3

2.2 Container Migration

The container migration is done through Podman's live migration API that consists of pre-dumps, dumps and restores. Pre-dumps are asynchronous checkpointing of the container state, which is used for initial checkpointing so the container keeps running and is not disrupted when the largest amount of data needs to be transferred. Subsequent pre-dumps take this checkpoint and add the changed state on top of it, where the change becomes smaller as more pre-dumps are done and thus take less time. Finally, a dump (or checkpoint) is done which adds the changes on this checkpoint and actually stops the container to do the change. This is followed by a restore on the target VM which takes the checkpoint from this mount, loads the container state and starts it on the target VM. Since the maximal container state is covered by the pre-dump, the actual container downtime for the migration is minimal, making it live-migration.

Our system, for each migration, uses a sequence of 10 pre-dumps done sequentially and spaced by a predefined time period. This is followed by the checkpoint operation. These checkpoints are done on an EFS volume that is connected to all the VMs. The restore operation takes the checkpoint from this common EFS volume. For both our checkpoint (which consists of the podman pre-dump and dump instructions) and restore functions, we expose a HTTP API that can be called for out controller on each target VM. Thus, when our scheduler returns the migration map, we go through each entry and call the checkpoint operation for the specific container ID by calling the API on the source VM and then asynchronously calling the restore API on the target VM for that container ID, letting it finish whenever the checkpoint is available. This finishes our migration operation.

3 Evaluation Methods and Results

We tested out the proposed system for a variety of workloads and measured the net down time of the container. This down time also includes the latency introduced by accessing files over AWS EFS. We do not present results without the file access latency since in the common use case of the system, containers are checkpointed so that they can be started in another VM and the newly created checkpointed file is never available on the file cache on other VM's in the cluster.

We used the `progrium/stress`[2] container to generate CPU and Memory bound workloads and all tests were performed on a cluster of AWS t2.micro instances.

3.1 CPU bound workload

To create a CPU bound workload we passed `--cpu 2` flag while running the `progrium/stress` container. This spawns two threads in the container, each of which computes the `sqrt()` of a number on an infinite loop. The observed average CPU and Memory workload as reported by `podman stats` is reported in the table.

CPU %	MEM %
99.74%	0.02%

The migration time for this container is presented in the following table.

Checkpoint(ms)	Restore(ms)
475.52	1110.44

We tested yet another CPU bound workload by passing `--io 3` flag to `progrium/stress` container. This creates three thread in the container which write to disk using `sync()` in an infinite loop. The migration times for this case were similar to those presented.
We also notice that that latency due to file accesses over EFS is a large percentage of overhead for restore latency.

3.2 Memory bound workload

To create a CPU bound workload we passed `--vm 1 --vm-bytes 256M --vm-hang 1 --vmkeep` flag while running the `progrium/stress` container. This spawns 1 thread in the container, which first allocated 256MB of memory using `malloc` and redirties the memory in an infinite loop with a sleep time of 1s on each loop. The observed average CPU and Memory workload as reported by `podman stats` is reported in the table.

CPU %	MEM %	MEM Usage
6.92%	26.28%	268.7MB / 1.022GB

The migration time for this container is presented in the following table.

Checkpoint(s) %	Restore(s) %
2.75	4.06

Due to limited available memory we were unable to increase the memory pressure at the level of granularity provided by `progrium/stress`. However, the reported values accurately reflect the expected result that memory copy presents a large overhead.

3.3 Mixed workload

We emulate a production workload by creating a stress test with both CPU and Memory bound process. We also include threads running `sync()` to emulate block I/O. We pass `--cpu 1 --io 1 --vm 2`

`--vm-bytes 128M --vm-hang 2 --vm-keep` flags to `progrium/stress` to achieve this effect. The observed average CPU and Memory workload as reported by `podman stats` is reported in the table.

CPU %	MEM %	MEM Usage
78.24%	26.30%	268.MB / 1.022GB

The migration time for this container is presented in the following table.

Checkpoint(s)	Restore(s)
2.99	5.06

4 Conclusion

The designed system allows for fairly transparent live migration of containers. The design decision to have heartbeats sent by the servers instead of the controller makes it easy to update the current VM map when a new VM is added to the cluster without any additional mechanism. Since, the checkpointed data is shared between servers using NFS, there is no need for servers to know the addresses of other servers in the cluster. This decision further improved transparency. It also allows for restoring decisions to be made lazily since virtually every server in the cluster has access to the checkpointed files. The current implementation does not provide any fault tolerance for either the controller or the servers, however, if the controller crashes there are no side effects to the cluster other than migration not being possible.

In all cases transfer of data through EFS presents a bottleneck and increases the restore time substantially. Performance could be improved by having another layer of caching such as a local shared high-bandwidth I/O EBS storage for migrating between servers placed in the same AWS region. We were unable to test this out due to both time and monetary constraints.

The time taken to checkpoint and restore a memory bound process takes substantially large time than even with using iterative checkpoints. The current fixed iteration number checkpoint certainly presents room for improvement and a possible optimization would be to iterate till the changed memory is below a certain fraction of the previous iteration or we have reached a predefined limit of iterations.

A possible feature addition which is not currently implemented would be a policy implementation for the master to be able to checkpoint and restore containers in servers based on global resource usage of the system so as to allow dynamic scaling of the size of the cluster. The mechanism to implement dynamic scaling is already in place.

Overall, the system presents a sound plan to load balance a group of containers on a fixed set of servers using live container migration. We have reduced network communication wherever possible and aimed for transparency.

References

[1] Podman. `https://docs.podman.io/en/latest/`.

[2] progrium/stress. `https://hub.docker.com/r/progrium/stress/`.

[3] runc. `https://github.com/opencontainers/runc`.

5 Appendix

5.1 Problems Faced

We faced multiple problems for live migration as well as metrics gathering in this project, requiring us to both pivot our goals and change our approaches to doing the project. Initially, we had tried to use CRIU support to the Kubernetes cluster management system and use that to enhance the cluster migration application with live migration. However, we ran into problems for supporting CRIU for every container supported by Kubernetes, which we have documented in our midterm report. After trying multiple cluster management systems on different OSes, we were finally able to run Podman's live migration API on Fedora-based linux distributions. We later realized that all of the issues we faced to get checkpointing to work were due to lack of syscall support needed by CRIU on the kernel version we were using on out server. Upgrading the kernel to v6.0.0 solved all out issues related to CRIU. We moved to a much cheaper t2.micro AMI running Debian to perform all our experiments.

Majority of the features we use in our base software, that is runc, Podman, CRIU were only supported in the latest releases of the software and we had to compile CRIU and runc on the server since the latest releases were not available via `apt` package manager.
We had then decided to use Prometheus to track Podman containers metrics as it was very extensive and comprehensive, making the load management much simpler. However, we ran into multiple problems with this for 2 weeks. Primarily, transporting files containing Prometheus metrics connected to containers inside another VM made us run into issues, even after configuring the networks, firewalls, etc. Finally, we decided into using the much more limited functionality of Podman Stats, with its limited streaming data to build our load balancing application.

Once we had our setup we spent more time on designing the system to be loosely coupled and avoiding unnecessary network traffic. While building the final application, we still ran into a few storage issues. We first tried connecting the cheaper EBS volumes to our EC2 instance cluster for storing checkpoint files, but EBS volume sharing was only enabled for higher storage and AMI tiers. We ran into permission issues attempting to upgrade to higher tiers. We finally decided to use EFS volumes as the networked storage for our cluster.