

Container Live Migration

Department of CSE, IIT Bombay

Submitted by :

Jayesh Kshirsagar : 153050013

Jayant Golhar : 153050062

Under the Guidance of :

Prof. Puru

Prof. Raju

1. Motivation

Containers are lightweight virtual environments. Main difference between container and traditional virtual environments is that containers make use of services provided by underlying Operating Systems, whereas traditional VMs have their own OS. Hence containers require considerably low resources compared to VMs. Live Migration is very old and well studied problem in virtualized environments. In the recent times, virtualization is mainly used in Large scale data centers, which consist of many commodity machines interconnected to provide many services to clients both within and outside the enterprise. In this scenario many VMs are created on the same physical machine so that proper isolation between various services is achieved. As one of the many added benefits, high availability can be provided via live migration.

In simple terms, live migration is migrating a VM from one physical host to other physical host, *without killing the VM*. That means all the state of the VM, in terms of execution state, opened files, active connections, etc. remains unchanged while VM is moved to destination and it resumes the execution from where it was stopped at source machine. Many further optimizations to this process are possible which aim to provide seamless experience to client while migration occurs.

Migration of VM is studied extensively and even provided as a production feature in many Virtualization products such as VMWare, Xen, etc. However containers is relatively young technology and many features provided by commercial VMM providers such as VMWare, Xen, etc. are not yet provided as production features. At the time of this writing, LXC(Linux Containers) is the base package for container technology on Linux and Docker (<https://www.docker.com/>) is the main commercial player that provides container technology based on LXC.

At the time of this writing, Docker(or even LXC for that matter) doesn't provide in-built live migration facility for containers. Given the growing interest in the container technology due to their light-weightedness, it will be good contribution to study and if possible implement the live migration of containers. This is the motivation behind the project.

2. Project Description and overall design

From high level, the problem we try to solve is this : A container(LXC or Docker) is running on a machine, we halt it's execution on this source machine and try to *resume it at the same state where it was halted on source*, on some other destination machine.

A straightforward approach to this problem is as follows:

- 1) Capture the state of container running on source machine.
- 2) Halt the container and transfer this state to destination.
- 3) Use this state information to resume the execution at destination.

This solution may result in temporary interruption to service being provided by container. However this interruption can be minimized, once this basic solution is achieved.

So it can be seen that main problem here is how to capture state of container so that it can be resumed afterwards without loss of state. During our research, we came to know about a project called CRIU(Checkpoint/Restore in Userspace: https://criu.org/Main_Page) which has successfully solved problem (not of migration) for lightweight processes. We therefore decided to use CRIU as a tool in our solution. A brief description of CRIU is as follows:

- **CRIU :** As the name suggests CRIU provides two main commands: 1) checkpoint and 2) restore. Using 'checkpoint' we capture the state of process provided as input to command into a collection of files. And 'restore' uses these files to resume a process. Although this works for a normal linux process, CRIU doesn't do it for LXC containers by default, and work is still going at both CRIU and LXC to integrate these two technologies with the same vision of using it for live migration(mainly, there are many other uses of this feature).

As base LXC package and CRIU have not yet been integrated successfully, we turned our direction of research towards 'Docker'. We found out that Docker also tried to use CRIU to provide 'checkpoint' feature, wherein a state of docker container along with state of processes inside it can be captured as a set of files to 'restore' the container along with it's processes. However this feature is still provided in the experimental branch of Docker and not has been released as a commercial feature (possibly due to some issues between Docker and CRIU integration.) Therefore we decided to use this experimental build of the Docker instead of commercial version.

If this docker checkpoint/restore feature works, then next step is to optimize the migration processs. Specifically, parameters to evaluate performance of migration process are: Total migration time and migration downtime, and they are defined as follows:

- Total migration time is time elapsed since the migration process started, till it finished.
- Migration Downtime is time for which container was unavailable due to migration process.

It must be noted that both above parameters have same value for stop-and-copy migration that we have considered till now. So from this point onwards our objective is to minimize the Migration downtime so that the migration process becomes seamless.

While above description seems fairly simplified, in each step of solution above, lot of configuration issues occurred. And resolving them requires very deep level knowledge of LXC/Docker/CRIU implementation, which we were not able to achieve in given short time frame. Therefore, after fair amount of self-efforts, we decided to re-use a pre-configured VM that supports Docker checkpoint/restore using CRIU. Details of this are provided in the implementation section.

However actual migration process still remained to be designed. To minimize the downtime we decided to use well known iterative live migration approach. Details of this approach, as well as final design of our project are as follows:

- 1) We start with a docker container running on source machine. We use experimental build of docker to use in-built checkpointing feature. In the initial pre-iteration stage, we checkpoint the container, but don't halt it and leave it running. And then we transfer the state obtained during the checkpoint to destination.
- 2) Once this transfer is complete, then we proceed to iterative transfer stage. This stage consists of several iterations. In each of this iterations, we checkpoint the container and compare the state obtained in this iteration with previous iteration's state and compute the difference between them. Now we send this difference, instead of complete state, to destination. And at destination we merge this difference with previous state to obtain new state that would have been transferred from source.
- 3) In the last iteration, we perform the similar process, but we stop the container running at source after checkpointing it. So the container remains down for the duration in which difference is computed, transferred to destination and the container is restored at destination. This duration is called the migration downtime.

In the following sections, we explain exact implementation details and experimental evaluation of our approach.

Implementation Details :

CRIU is an ongoing project and developers are facing some problem with configuring CRIU with LXC. We tried to configure CRIU and LXC but faced some issues like “SECCOM filter not supported” which the developers are currently trying to solve. So we thought of using Docker for our project but CRIU checkpointing feature is still not available in stable version of docker and it is in experimental phase. So we thought of using experimental version of docker with checkpointing feature. One developer has already configured this docker and CRIU as an experimental unit and provided this bundle using vagrant box. So finally we are using Vagrant box which can be hosted on our machine and vagrant box contains docker's experimental build which we can use to create container, checkpoint container, etc.

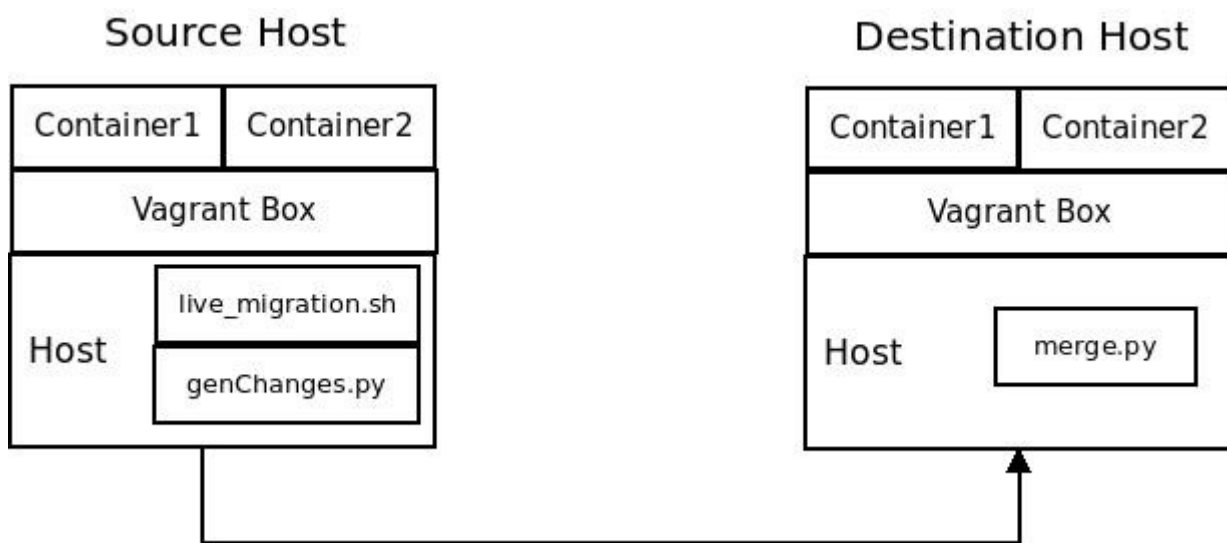


Figure 1: Live-migration Process

So there are two hosts, one acts as source and another acts as destination for live-migration process. Each host is running vagrant box and container is running inside vagrant box, which we need to migrate from source to destination.

We are running bash script “live-migration.sh” at source for live-migration process which does the following work sequentially. We referred the following source to develop our script (<http://blog.circleci.com/checkpoint-and-restore-docker-container-with-criu/>).

1. There are three arguments to the “**live-migration.sh**” script viz. a) Name of container, b) source vagrant box name c) destination vagrant box name.
2. There are multiple rounds in iterative live-migration process. In each round, we are transferring only that state of container which gets changed after the previous round, except for the first round.
3. In the first round, we will checkpoint the container and keep it running. This checkpointed state of container get saved in file system of Vagrant .

We will fetch this saved state from vagrant box to local source machine using **scp**.

4. Then we will **ssh** the destination host and transfer this complete saved state to destination host using **scp**.
5. After this we will run the iterative migration process for remaining rounds which can be halted after a predefined number of rounds or using some heuristic which we plan to develop as future work. (In fact, we have developed initial version of this heuristic, but we were not able to test it properly due to time constraints.)
6. In each round, we will checkpoint the container and keep it running till we don't hit the last round.
7. We then copy this checkpointed state from vagrant box to source machine which is then compared with previous checkpointed state using python script **“genChanges.py”**
8. In fact changes are of two types, one is when some changes has been added in the available file from previous checkpointed state and another type is of newly added files in current checkpointed state other than previous state.
9. We send these two changes to the destination machine using ssh and scp.
10. After receiving the changes, we merge this changes to received container state at destination using python script **“merge.py”**
11. After last round, we need to restore the container in destination vagrant box from the received container state. We use 'docker restore' command at destination vagrant box to resume the container from checkpointed state(of course this process is automated).

Script on source host : live_migration.sh, genChanges.py

Script on destination host : merge.py

File structure of changes file generated by **“genChanges.py”** :

Basically there are three sections :

1. Replace : **“replace”** word followed by series of line containing **“byte number to change”** and **“changed byte value”**
2. Truncate : **“truncate”** word followed by size of the file to truncate to.
3. Append : **“append”** word followed by series of **“changed byte value”**

Heuristic design to terminate the iterative process :

There are two criteria that we are using to halt the iterative process and start final stop-and-copy round.

1. First we compare the size of the current change state to previous round change state and if current change state size is greater then terminate the iterative process.
2. Second we check if current change state size is less than some threshold value and if it is less then terminate the iterative process. This threshold value is configurable.

Experimental evaluation:

Our evaluation was aimed to answer following questions about software that we developed:

- 1) Correctness: Is software working correctly?
- 2) Sensitivity analysis: This part was pretty open ended. We decided upon some parameters and then varied them and studied their impact on migration downtime, as this is the most visible attribute of the migration process. This is explained in detail below.

Correctness:

Initially we were able to migrate a docker containing simple process which was printing numbers sequentially. To test it further, we designed incrementally complicated processes, namely process generating fibonacci series and process generating prime numbers. As the computation becomes more involved, more resources get associated with process and this results into complicating representation of the state of container. So the idea was to test, whether container gets restored in the exact state at the destination. In our fibonacci and prime generator processes, it means that process should start printing next number in the sequence at destination.

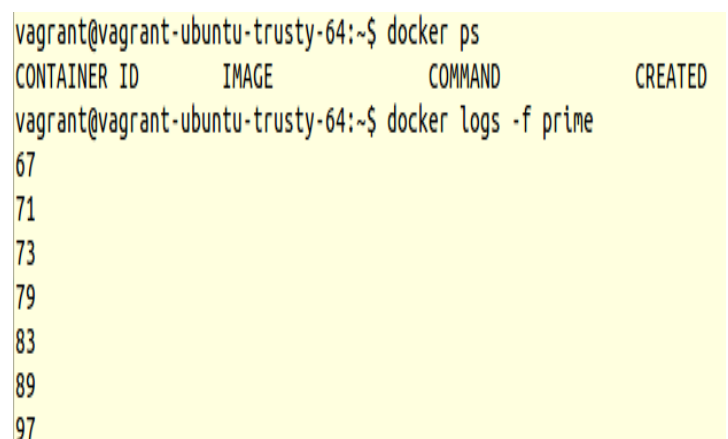
We migrated containers running these processes at various stages of their execution and our migration process was able to restore them in the exact required state.

Following figures show one of the live migration achieved by our solution.:



```
vagrant@vagrant-ubuntu-trusty-64: ~  
^C  
vagrant@vagrant-ubuntu-trusty-64:~$ docker rm -f mem  
Error response from daemon: no such id: mem  
Error: failed to remove containers: [mem]  
vagrant@vagrant-ubuntu-trusty-64:~$ bash prime.sh  
prime  
3  
5  
7  
11  
13  
17  
19  
23  
29  
31  
37  
41  
43  
47  
53  
59  
61
```

Illustration 2: Prime Stops at 61 on source and resumes at 67 on destination



```
vagrant@vagrant-ubuntu-trusty-64:~$ docker ps  
CONTAINER ID        IMAGE               COMMAND             CREATED  
vagrant@vagrant-ubuntu-trusty-64:~$ docker logs -f prime  
67  
71  
73  
79  
83  
89  
97
```

Sensitivity Analysis:

To observe the interdependence of various parameters in the migration process, we created a docker container containing a process with following specifications.

The objective of this process was to mimic execution of any process in practice. Any normal process reserves some amount of memory initially, randomly reads data from this memory, performs some computation on data and then repeats this process.

To mimic this behaviour, our dummy process reserves array of given size. Then it decides number of elements to be accessed in the array(memory access) and their locations in array *randomly*, then it accesses these position in array. Then process sleeps for a random amount of time and performs similar actions in further iterations.

For e.g., In a particular iteration, process may decide to touch 4 positions and these positions may be 23, 17, 34, 99 in an array of size 100. The randomization in each action mimics the practical behaviour of a process, wherein it is not possible to predict the exact process behaviour.

This process has following main configurable parameters:

- 1) Array size: this can be taken as memory consumption of process.
- 2) Speed at which memory is accessed: This parameter has a significant impact on iterative migration process.

Using this parameters we desinged experiments as follows:

Experiment 1: Determining impact of memory consumption on migration downtime.

In this experiment we investigated how migration downtime changes as the memory consumption of process in the container changes. The size of array in above process was taken as memory consumption of process. We increased the size of array within a range and measured corresponding downtimes. The observations obtained are as follows:

Array size(Memory consumption)	Migration downtime(in seconds)	Total migration time
100	14	41
200	13	38
400	14	37
800	14	40
1600	15	44
3200	15	45
12800	15	44
25600	20	69
51200	24	76
102400	32	92

Table 1: Observations for Experiment 1

It can be seen that, our downtime is in seconds for such a small process while commercial solutions provide downtimes of milliseconds. Many factors, such as availability of less bandwidth during experimentation, less-than-optimal implmentation for computing and representing the difference between checkpoints, are reason for this. Our aim in this project was mainly implementing the process of live migration correctly. During our experimentation we have found out many possible optimizations to overall process, which can be seen as the future work of this project.

From above data and plot, we can conclude that for smaller sizes of memory downtime remains almost same. But as the size grows exponentially, the downtime also increases.

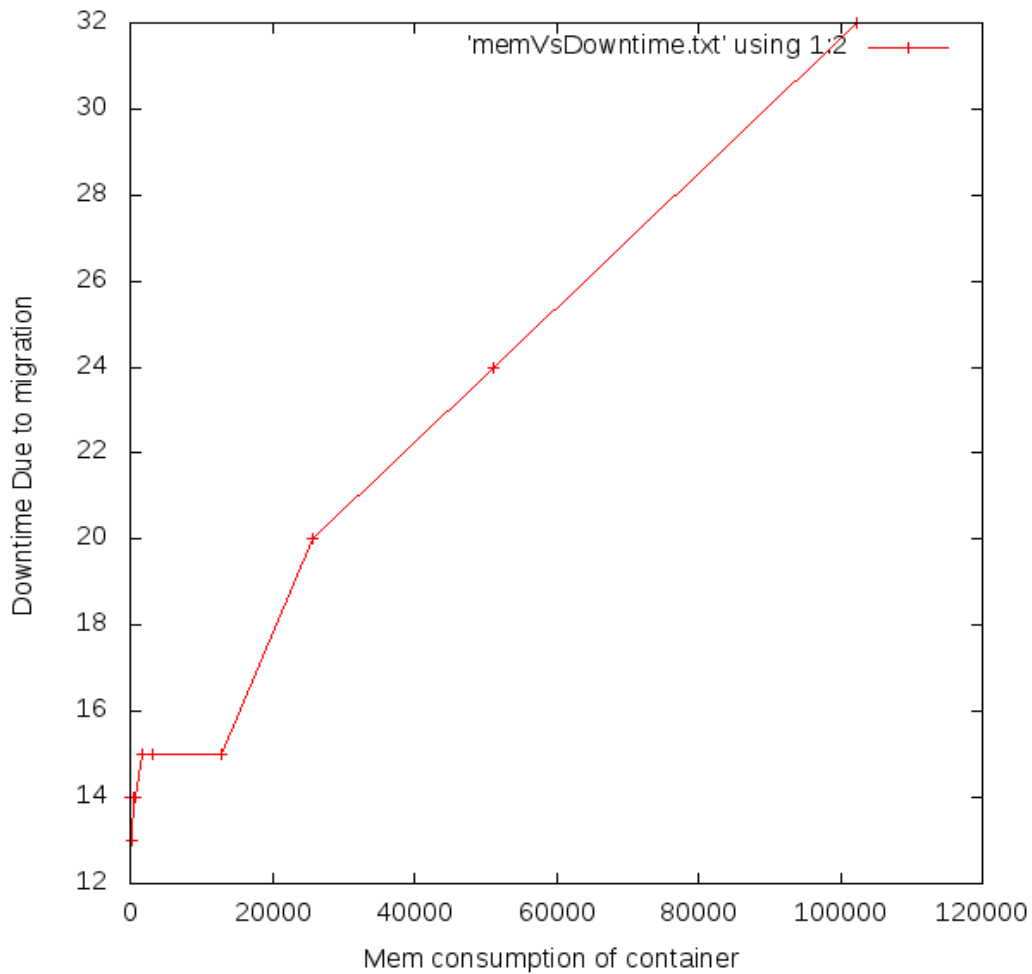


Illustration 3: Plot of data in Table 1

Experiment 2: Analyzing impact of memory access speed on downtime.

As explained above, we send the difference between checkpoints repeatedly while container executes. If memory access speed is high, the size of difference between consecutive checkpoints will be larger, impacting the downtime. Our dummy process sleeps between iterations. We can increase memory access speed by reducing sleep duration between iterations and increasing the amount of memory accessed in each iteration. That means the delay for which process sleeps between iterations is inversely proportional to memory access speed. So we computed an approximate indicator of memory access speed as a function of a configurable parameter 'max_delay' for our process, and this function is: $(1/\text{max_delay}) * \text{constant}$. We took constant=50000, because it gives good visualization of the observations. Then in series of experiments we reduced the 'max_delay' parameter in our process, thereby increasing memory access speed and observed the migration downtime.

Observations are as follows:

Max_delay	Speed	Migration Downtime(seconds)
32000	1.15625	15
16000	3.125	16
8000	6.25	15
4000	12.5	16
2000	25	18
1000	50	17
500	100	17
100	500	21
50	1000	25

Table 2: Impact of memory access speed on downtime

The plot of this data is given below:

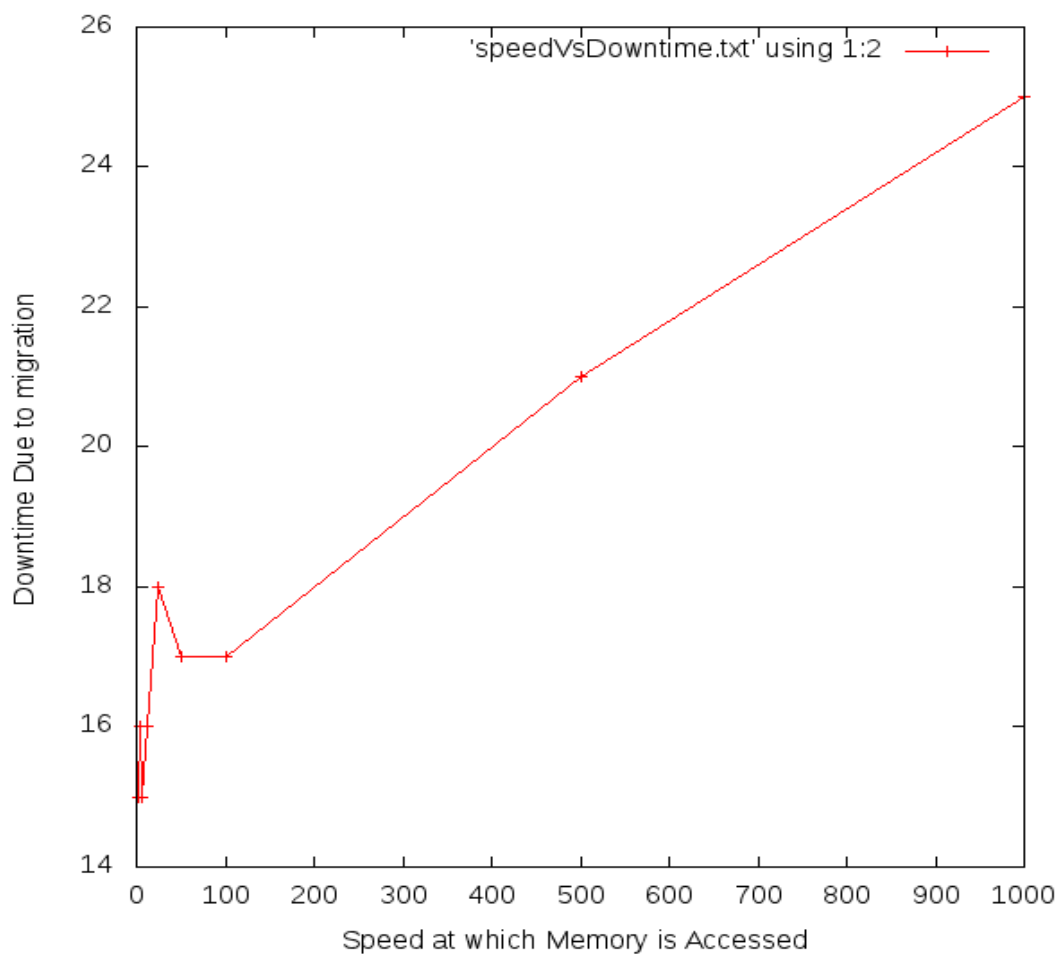


Illustration 4: Plot of data in Table 2

From above observations and graph, we can see that, as memory access speed increases, the migration downtime also increases.

Conclusion:

In this project, we investigated the problem of Container Live Migration. We were able to develop a solution based on iterative live migration strategy and ran several experiments which gave us fair idea of various aspects of our solution.

This project gave us good idea of how the problem is formulated, how are various strategies to tackle a given problem designed, and how to design various experiments to test the proposed methodology. In conclusion we can say that, we have learned a lot in this process and in entire course. We look forward to study further topics in this excellent area.