Live Pod Migration in Kubernetes

Henry Loo, Andrea Yeo, Kelvin Yip, Thomas Liu

Introduction

Live migration is a pervasive technique in the realm of virtual machines, allowing transparent movement of VMs from one physical machine to another with negligible service disruption (hence the term live). Recently, there has been increasing interest to apply the same technique to containers.

Kubernetes, Google's open source project for cluster orchestration, does not presently support live migration of its pods, the system's units of organization. There have been discussions among the project maintainers around live pod migration, but to our knowledge the feature is not actively being worked on. In our project, we make an experimental effort to implement live pod migration in Kubernetes, building atop of the recently introduced possibility of Docker container migration.

Background

With recent trends toward scalability and distributed microservices, containerization has quickly become a lightweight and widely adopted alternative to VMs or physical nodes as the unit of deployment. A traditional setup of one service per VM or per physical machine provides strong isolation, but comes with a large amount of overhead. Although two independent services do require some level of isolation, having two copies of the entire OS or hardware is not an efficient use of resources. Containers implement the minimal needed process isolation for running services, while sharing the same underlying OS and hardware infrastructure.

Although containers have been in use for many years with major data center deployments such as those at Google and Facebook, Docker can be credited for the widespread popularization of containerization. Building on top of Linux containerization primitives, Docker provides a much more accessible interface and ecosystem for packaging and deploying applications.

The biggest benefit of containers, that they can be quickly created and destroyed in large numbers, is also one of their biggest challenges. Because a large number of containers can share one set of resources (e.g. CPU, RAM, disk) with no individual container confines, containers require careful orchestration to be used effectively. Orchestration responsibilities include scheduling/packing containers onto machines, detecting and reacting to failed containers, and maintaining constraints such as number and location of replicas or related containers. Well-known systems related to orchestration include Mesos, Aurora, Borg (internal at Google), Tupperware (internal at Facebook), and most recently Kubernetes from Google.

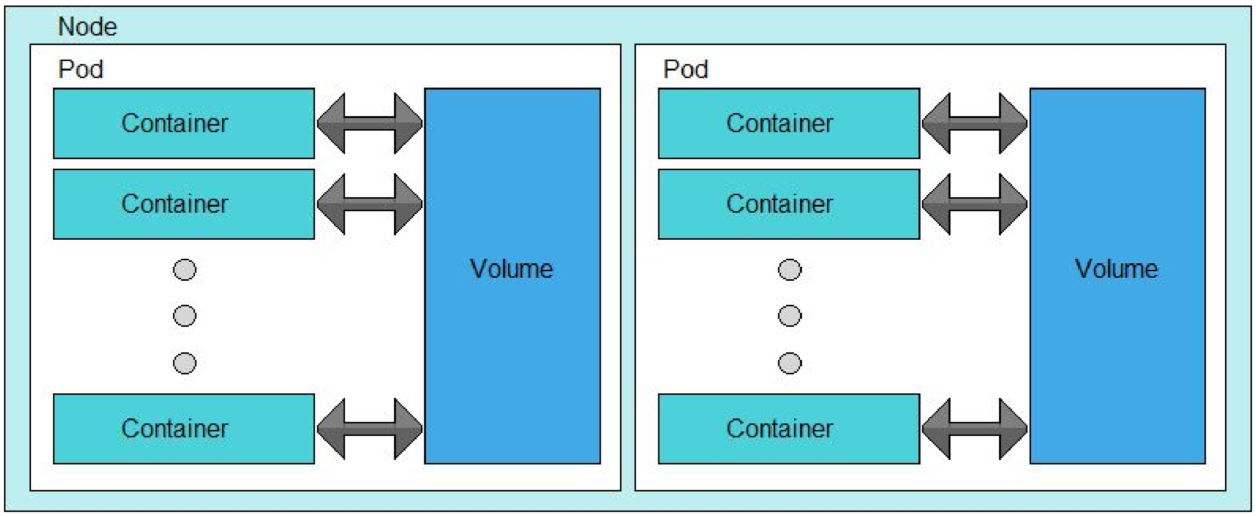

Figure 1: Layout of containers, volumes, pods, and nodes in Kubernetes

Kubernetes is an open source orchestration system that takes inspiration from the design of Google's internal system, Borg. In Kubernetes, related containers are organized together as a pod, within which they share an IP address, port number space, and intra-pod file systems known as volumes. Containers within the same pod can communicate to each other by localhost addressing or typical IPC methods. Containers across pods can only communicate using network, for which Kubernetes provides a service discovery mechanism. Pods can be assigned "labels" so that they can be easily identified and queried. Pods and labels together form logical and distinguishable units with which Kubernetes can work with and schedule onto server nodes.

Problem—Relocation of Pods

Many scenarios would benefit from the ability to relocate active pods. To list a few, we briefly summarize two active Kubernetes proposals that may involve pod relocation:

- If, for example, we wanted to replace an unreliable or under-resourced node, a cold swap may cause service disruptions. Instead, we wish to mark the node as decommissioned and gradually phase it out. In Kubernetes, this should indicate that no new pods are allowed to be scheduled to the decommissioned node, and any existing pods on the node should be relocated as soon as possible.

- The ability to move a running service or pod between data centers or cloud infrastructure providers would be desirable for many reasons:
    - When there are major disruptions to a data center or infrastructure provider, we can perform an 'evacuation' of services to another location so that users are not impacted.
    - We may be motivated to dynamically switch node locations for cost-effectiveness. Maybe one infrastructure provider is cheaper than another during a certain time period or certain level of activity.

  There is a major effort currently happening around cross-cluster federation, wittingly nicknamed 'Ubernetes', to allow for this sort of communication and collaboration.

Currently, relocating a pod in Kubernetes is only possible by disposing of the source pod, then recreating a new pod of the same type/template from scratch. Application developers need to design around this fact by not relying on longevity of pod-local state and by storing any necessary data into pod-independent persistent storage, such as an external database. Coupled with sufficient replication of services, it is certainly well within reach to architect zero downtime systems even when pods are restarted and relocated in this manner. However, support for live migration of pod state would move this burden down to the orchestration level. With highly stateful services, in-memory caches being one example, this would be a major convenience to application developers.

Approach—Live Pod Migration

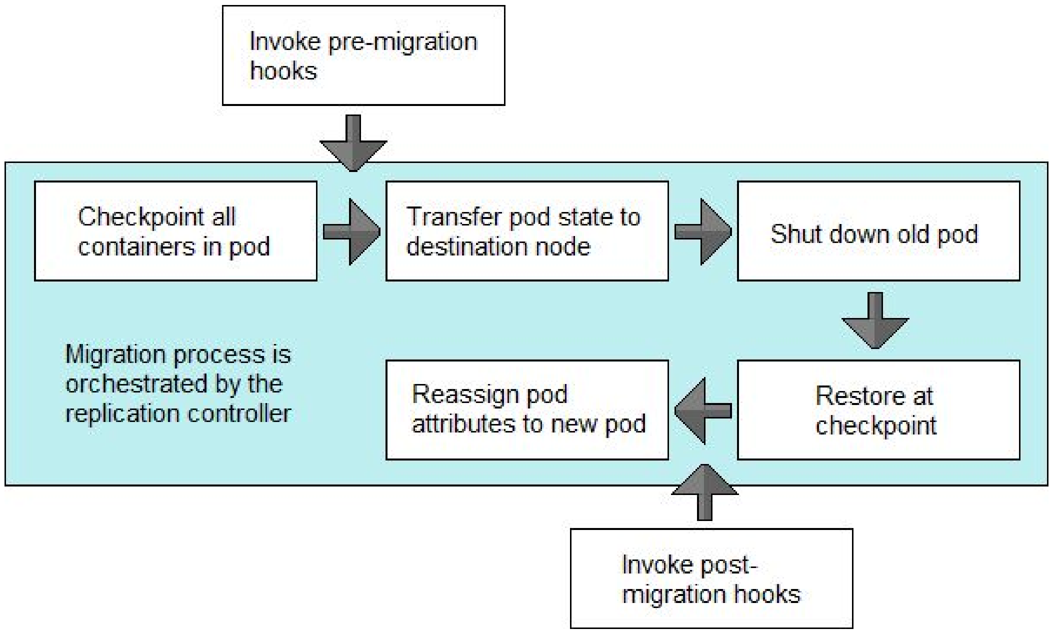
Invoke pre-migration hooks
Checkpoint all containers in pod
Transfer pod state to destination node
Shut down old pod
Migration process is orchestrated by the replication controller
Reassign pod attributes to new pod
Restore at checkpoint
Invoke post-migration hooks

Figure 2: Overview of this project's approach to the migration process

To achieve the goal of transparently relocating a pod while preserving all pod state with minimal service disruption, there are a number of subproblems we need to address. We briefly enumerate these problems and their potential solutions on a high level, in no particular order.

Capturing Container State

A pod consists of a collection of Docker containers—each of these containers must have their states transferred from the source pod to destination pod. A requirement for this is to have a reliable and efficient way to live capture container state into a transmittable format. The CRIU (Checkpoint Restore in Userspace) project aims to solve the aforementioned problem for active Linux processes. CRIU can be used alongside Docker, but native integration that provides a more seamless result is in the works. In fact, there is a non-upstream (some rework is being done for compatibility with another Docker update) Docker fork that contains this feature in functional state, with which we can build binaries from for the purpose of this project.

Checkpoint restore addresses container migration. This process is performed in several steps:

1) Freeze a running application
2) Create a checkpoint for the address space and state of the process tree
3) Restore the checkpointed process tree
4) Resume the application from its last frozen point

In Kubernetes, pod mutation is orchestrated by the processes on the master node(s). Specifically, schedulers assign pods to nodes and replication controllers communicate with the appropriate nodes to create, delete, or update pods. The low level work of maintaining pods and creating containers becomes the responsibility of the Kubernetes agents, known as Kubelets, resident on the target nodes themselves.

Kubelets currently interface with Docker to pull container images, start and stop them, and monitor their lifecycles. Expectedly given the alpha status of Docker checkpoint/restore, no functions exist on the Kubelet to perform a state capture operation and to export the data. We list the Kubelet modifications needed to support the checkpoint/restore operations below, and in the next section describe necessary changes for external interactions, namely API allowing for data transmission and orchestration of this process.

1) Implement a helper function to invoke the checkpoint Docker command—this action causes a dump of multiple image files representing the live state of a container. Locate this these files and return file handles for further processing.
2) Implement the inverse operation, invoking the restore Docker command—given the images representing container state, relaunches the container in identical state to time of checkpoint.

A number of concerns must be attended to when integrating checkpoint/restore, related to downtime and failures.

1) The checkpoint operation is intended to halt the container until restored, either locally or remotely. We have two options we can take to design around this.
    a) Live migration with VMs typically involve multiple stages to mitigate downtime, transferring a reasonably consistent representation of state while the source machine continues to operate. It then has a post-transfer stage where any 'dirtied' state, or memory pages in the context of VMs, since the original transfer are corrected. During the post-transfer stage, the target VM is now active, while the source VM is decommissioned, only alive until completion of the un-dirtying process. It may take further investigation to see whether a similar approach can be taken with checkpoint/restore, where we immediately resume a container following a checkpoint on it. That way the container continues operating until its replacement is ready to take its place. Factors affecting feasibility of this approach include:
        i) Existence of a method to track and transfer post-checkpoint state changes: If a solution requires a significant amount of work and possibly modification to Docker, it will not be possible within the timeframe of our project.
        ii) Latency of checkpoint image transfer: if the transfer of actual state data takes a significant amount of time, post-checkpoint state changes may be too significant to correct.
    b) Alternatively, the above approach may not be feasible or even necessary, in which case we simply freeze the container and its resident pod after checkpoint, assuming that existence of replica containers or pods will mitigate service disruption from this freeze.

Transportation of Pod State

Having a pod's container checkpoint images, we also need a mechanism for transportation of this data between nodes. The only current example of internodal data communication within Kubernetes is the transfer of small configuration data, such as pod templates. Furthermore, nodes do not currently communicate with each other in the orchestration layer. To make state migration possible, we need a routine on the Kubelet level that does the following:

1) Checkpoint all containers, respecting any dependency ordering between containers, within the source pod/node.
2) Establishes a communication channel with a remote destination node, with the address known from input.
3) Gradually transmit the data to the destination node.
4) Receive data until the entire set of checkpoint images is received.
5) Restore each of the containers, respecting dependency ordering.
6) Once migration is complete, delete the pod and any containers on the source node.

7) If communication failure happens mid-transfer, we attempt to retry a number of times. In case of irrecoverable failure, we simply abandon the process and the old pod continues to operate as normal. It is not the Kubelet's responsibility to ensure eventual success of migration.

In this project, we don't intend to handle complex recovery from corner case failures. Additionally, the transportation process has the same assumptions as the general live migration process-both pods and their residing nodes must be healthy until migration completion, otherwise we simply retry or fail the transaction.

In addition to container state, shared pod volumes must also be migrated using a similar data transfer mechanism. On a high level, this should be a simpler transfer than running state since we can simply treat it as a directory copy.

Reassignment of Addressable Properties

When a pod is relocated, any addressable attributes such as the hostname, IP address, and active ports must be identical between the source and destination pods. Non-addressing fields such as Kubernetes-internal IDs and names are not transferred. To make a special note on preservation of ports, Kubernetes gives each pod its own port space, so migration should never cause port number contention with other existing pods.

At the moment, there is no mechanism for Kubernetes to transfer an IP address between pods. Rather, each node is assigned a fixed IP address range that its pods can use. Maintainers suggest implementing a network controller that holds the logic for this process, just as the replication controller will be responsible for the transfer of pod state. We intend to write a fairly simple prototype of a network controller that can dynamically allocate available addresses out of an IP address pool.

Orchestration and Coordination of Migration

The replication controller on the master node(s) of Kubernetes is currently responsible for maintaining the lifecycle of pods, creating or destroying pods to ensure that there is always a specified number of replicas. We need to extend the controller to also coordinate the migration effort between two nodes. Given that Kubelets on nodes already have an interface to initiate checkpoint, restore, and state transfer, the replication controller has the following responsibilities:

1) Initiating the pod snapshotting and transfer process on the source and destination nodes.
2) Making sure no requests are prematurely routed to the pod on the destination node until migration is complete, similar to during creation of a new pod.

3) Monitoring the migration process, detecting failures or completion of state transfer.
4) Once the destination pod is in runnable state, the replication controller invokes the network service described above to finally migrate the addressable of the old pod to the new one, officially redirecting all traffic to the new pod. The old pod is then discarded. There may be a slight service disruption during this final transitional phase.

External Awareness of Migration Events

A side requirement, as mentioned in the maintainer discussion, is the ability for external services to be aware of ongoing migrations. Components that have been affected by a completed migration should be able to react accordingly. To handle this, we must implement pre- and post-migration hooks that external services or modules can use to perform any necessary reactions, such as retrieving the new pod's name for logging purposes. Kubernetes already provides lifecycle hooks for creation and deletion that we can model our migration hooks off of.

CLI Migration Commands

One last convenience requirement that should not drastically affect our project scope is the inclusion of a CLI, known as kubectl, command to initiate pod migration. There is extensive documentation on CLI conventions and many source code examples for how to tie a command to an internal API call.

Assumptions and Constraints

Although we approach a large and general problem as described in the next section, we identify a number of assumptions and operational constraints that our solution should work in. This allows us to keep the project manageable, whereas a comprehensive solution would not be in scope for a month-long effort.

1) Live pod migration should be performed in a stable environment. We do not attempt to guarantee eventual success if there are node or network failures during the migration process.
2) We currently intend migration to be a manually invoked operation, and not reactive to certain events within the system.
3) We limit our test cases to pods containing typical web service components, such as an in-memory database like Redis, static HTTP content server, or simple JSON-responding services that may access communicate with pod-local data stores, but no external data stores. To be precise, we wish to keep all of the pod's state dependencies internal to

the pod itself to simplify our implementation task. However, external pod dependencies is still permitted as long as communication uses a generally stateless protocol, such as RESTful HTTP.

4) It is not in scope for our migration process to account for complex or mutual dependencies between containers during startup or restoration. Containers should be checkpointed and restored in the same order as that of pod creation/deletion.

Timeline

Since the first milestone (project proposal) is due on Monday, February 29th, the following milestones will also lie on Mondays (to allow for weekly progress updates).

Deadline	Task
Feb 29	Submit project proposal
Mar 7	Read and understand Kubernetes architecture documentation. Learn about Docker and its API.
Mar 14	Implement transmission of pod state between nodes
Mar 18	Schedule a project status meeting with TA
Mar 21	Implement network controller to handle dynamic reassignment of addressable properties within the cluster
Mar 28	Implement: Replication controller side migration logic Pre/post migration hooks and any necessary event emission Ability to initiate migration from CLI
Apr 4	Perform testing Add compatibility with ShiViz
Apr 11	Submit project code and final reports

SWOT Analysis

The following details will outline the internal (strengths/weaknesses) and external (threats/opportunities) constraints for this project's team.

Internals

Strengths:

- Team members have worked with each other on previous assignments for this course.
- Team members are diligent and punctual.
- Team members have experience and context with typical use cases for a cloud infrastructure like this (i.e. web application and API development).

Weaknesses:

- Some team members have minimal working experience with Go. Given that the project will be primarily using Go, this could slow progress.
- No team members have worked on Kubernetes before.
- Team members do not have experience working on significant open source projects and knowledge of best practices and workflow.

Externals

Opportunities:

- Kubernetes provides extensive user documentation to refer to.
- There is a relatively large community surrounding Kubernetes, including a live chat and active GitHub users. We can potentially ask for any guidance if needed.
- Kubernetes has strict coding standards and requirements for code commenting, making it easier for us to dig through source code.

Threats:

- UBC Science Co-op program may request mandatory job interviews that could result in meeting time conflicts for team members.
- Commitment to exams or assignments from other courses may interfere with progress.
- There isn't extensive documentation on the actual implementation and code organization of Kubernetes; contributors are generally expected to learn from source code.