k8s-live-migration

BUILDING A CONTAINER-FRIENDLY TELEPORTATION SYSTEM

April 11, 2016

Andrea Yeo	v5v8	34296129
Henry Loo	m6r8	33569120
Kelvin Yip	s8u8	18016121
Thomas Liu	y6x8	47867130

Abstract

Planned outages of online services happen for many reasons, ranging from routine infrastructure upgrades to forecasted natural disasters. When they do happen, live migration is an invaluable technique that can be used to move software, while it remains continuously running, from one machine or datacenter to another, and achieve near-to-zero service disruption. With virtual machines in particular, live migration is a heavily researched topic, and results are already being used in large scale production environments such as that of Google Compute Engine.

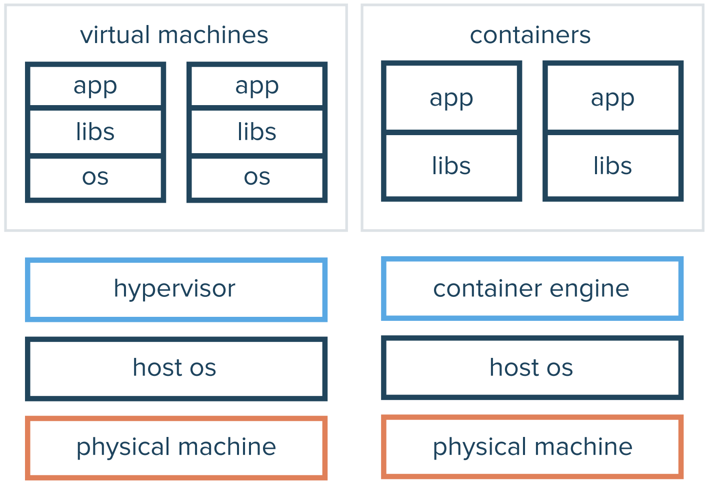
Recently, there has been an increasing amount of interest in applying the same technique to containers, a computing abstraction that carries many of the same benefits as virtual machines do, but often has significantly less overhead in resource usage. In this paper, we describe our effort to extend one container orchestration system, Kubernetes, with the ability to perform live migrations. Our project takes advantage of the experimental checkpoint/restore operations of Docker, and in doing so provides a prototype mechanism within Kubernetes to migrate the in-memory state, temporary disk volumes, IP addresses, and service discovery metadata of a set of containers, known as a pod, from one node to another, effectively a container-friendly teleportation system (no, it is not cat- or human-friendly quite yet).

Introduction

To keep this paper self-contained for readers who may not be familiar with the involved topics, we begin by providing background on the preexisting systems and concepts that our project builds atop of—their use cases, significances, and general philosophies. These sections can be skipped for readers who are already familiar with the involved ideas. We then proceed to motivate the subject of our project, live migration of pods in Kubernetes.

CONTAINERIZATION VS. VIRTUALIZATION

With the recent trend toward massively scalable and distributed microservices, containers have rapidly become a lightweight and widely adopted alternative to virtual machines as the unit of deployment. Both containerization and virtualization provide much-needed levels of flexibility and abstraction when used atop of physical servers. They allow for sets of applications with widely differing resource requirements, running times, and (possibly conflicting) software dependencies to run on the same physical machine. When used correctly, containerization and virtualization can dramatically reduce infrastructure-related costs and frustrations. The primary difference between the two is that, whereas each virtual machine runs its own isolated operating system, all containers share the same host operating system, and instead opt to provide isolation on the level of applications and their dependencies—libraries, file systems, and networks.



Virtualization *Containerization*

Container isolation guarantees, given the sharing of the host OS, are naturally weaker than those provided by virtual machines especially when concerning security. Containers also require applications to have roughly comparable operating system requirements. For example, while it is possible to run a Ubuntu- and a Gentoo-based container on the same host machine, the same cannot be said for running a Windows Server container alongside one that is Linux-based. However, in controlled and non-hostile deployment environments such as internal company datacenters, the performance benefits and resource savings of containers (from having many containers per OS as opposed to one OS per VM) often outweigh the stronger isolation that VMs provide.

In their raw Linux forms, containers have been used to great success for nearly a decade now within large company datacenters like those of Google and Facebook. However, their mainstream adoption can more or less be attributed to the emergence of Docker in 2013. Docker is a containerization system that uses native Linux containerization primitives under the hood, but manages to make containers much more approachable to application developers and operations engineers through a multitude of features. Some of these features include a commit-based script format and online registry system (analogous to git and Github) for creating, collaborating on, and distributing container images, both a CLI and REST-like HTTP API for managing and monitoring containers, and a network model that simplifies container deployment in the context of web services. Docker is the primary underlying containerization system that Kubernetes (and consequently our project) works with.

ORCHESTRATION OF CONTAINERS

To no surprise, managing containers on the scale of thousands is impractical without automation through software. Container orchestration systems have been designed specifically for this purpose, with responsibilities that include:

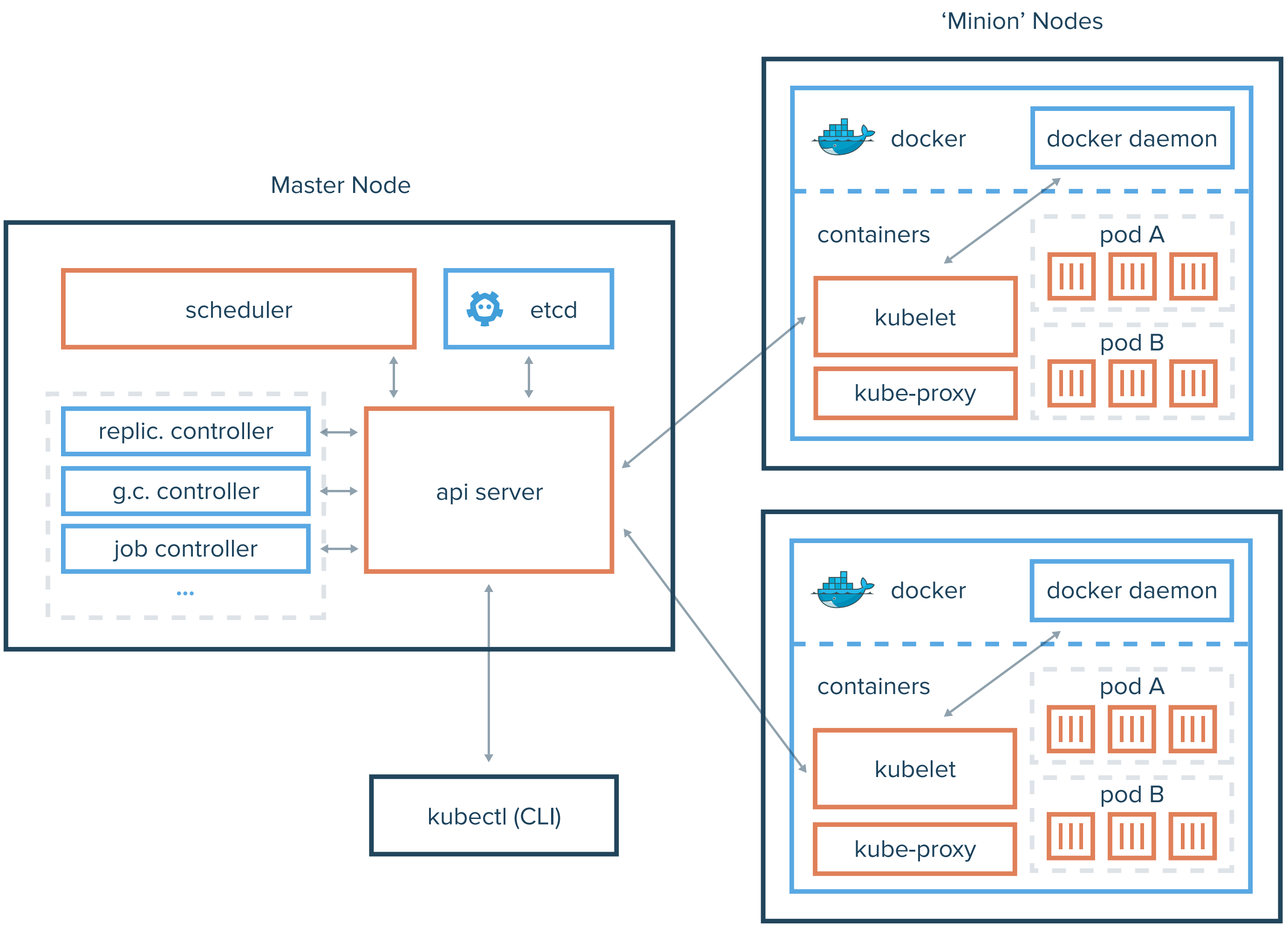
- Scheduling/packing containers onto servers based on resource requirements (e.g. CPU, RAM, disk) and running time
- Detecting and gracefully handling container and node failures
- Maintaining system-specific requirements such as the number of replicas (and where those replicas are located) that must be running at any given time for a particular service
- Routing service requests to the appropriate nodes and containers

Well-known orchestration systems include Borg (internal to Google), Tupperware (internal to Facebook), Apache Mesos and Aurora, and most recently, Kubernetes, open-sourced by Google in 2014.

OVERVIEW OF KUBERNETES

Kubernetes is an open source container orchestration system that takes inspiration from the design of Google's internal system, Borg. In Kubernetes, the logical unit of deployment is the pod, a group of closely related containers (typically representing one service) that share a network namespace and optionally, disk volumes. Containers within the same pod share the pod-scoped IP address and port number space, and can communicate with each other via localhost addressing (a huge convenience to application developers). Conversely, two different pods, whether or not they are running on the same machine, have their own separate network namespaces by default. Another thing to note is that all pods in Kubernetes are externally addressable, reachable by both other pods and external clients.

One node in a Kubernetes cluster is designated the master, responsible for keeping track of all nodes and pods within the cluster, and driving the cluster state toward the desired state specified by the Kubernetes user or administrator (how many of each type of pod should be running, what containers should be running within each pod, etc.). All other 'minion' nodes watch the master for changes to its pod assignments, and runs, kills, and updates pods accordingly.



Kubernetes Architecture

KUBERNETES MASTER NODE

The master node, though treated as one logical member in the cluster, comprises a number of independently running daemons each with different roles—the scheduler, etcd key-value store, API server, and controller manager. Our project does not involve the scheduler (the service responsible for matching pods to nodes based on resource requirements) and as such we only introduce the latter three components in the subsections below.

etcd

etcd is the distributed key-value store used by the master node to maintain all cluster state. By persisting to disk and using Raft for consensus, *etcd* provides the underlying high availability and reliability guarantees that make Kubernetes resilient to master node failures. Worthy of note is that it is entirely possible to cluster the *etcd* store rather than run it as a single instance on the master node. Most obviously, the master node's *etcd* instance contains a set of objects to represent all pods within the cluster, including their specifications and current statuses. Additionally, it contains objects representing nodes and controller specifications, the latter of which we describe alongside the controller manager.

API Server

The API server is a RESTful HTTP server that all Kubernetes components as well as external clients talk to in order to perform CRUD-like operations on cluster state. In additional to typical CRUD operations, the API server also provides a watching mechanism, where its clients can ask to be notified of changes on a particular subset of resources without resorting to inefficient polling. Clients of the API server include:

- External clients and users, who send requests to the API server to read and write pods and controller specifications
- Kubelets, which watch for changes to its pod assignment and update statuses of their nodes and pods
- Controllers, which watch for controller spec changes, read and write pods, and update statuses of their progresses

Controller Manager

The 'controller manager' is essentially all of the controllers in Kubernetes bundled together as one binary for convenience. Controllers contain the heart of the cluster orchestration logic—each controller drives the cluster's pod state toward some particular goal by interfacing API server as described in the previous section. These goals are either inherit, as in the case of the garbage collection controller that cleans up pod objects of no-longer-running pods, or driven by user-defined specs. As an example of the latter, the replication controller, one of the most fundamental controllers in Kubernetes, is spec-driven. Through the API server, users create replication specs, sent as JSON- or YAML-formatted text, that most importantly describe the number of desired replicas for particular pods. By watching the API server for changes to the replication specs set, the replication controller notices the newly created spec. It then continually monitors, creates, and deletes pods in order to maintain the desired replica count.

KUBERNETES 'MINION' NODE

All other nodes run the Docker daemon and two Kubernetes daemons—kube-proxy and kubelet (these are often run within Docker containers). The Docker daemon is the underlying container engine for the node, the kube-proxy performs request forwarding to pods resident on a node (a component that our project does not involve), and the kubelet performs the primary coordination with Docker and the master node, which we briefly discuss below.

Kubelet and the Sync Loop

A kubelet's primary routine is a synchronization loop that continually receives and acts on pod and container updates from both the master API server and the Docker daemon. In colloquial terms, the kubelet asks the master node, "what pods should I be running right now?", interfaces Docker to run, stop, and inspect statuses of containers as necessary, and periodically sends status updates for its pods to the master node. For example, if a pod or a container within a pod on the node crashes, the kubelet notifies the master node of the condition immediately and will by default restart any affected containers.

LIVE POD MIGRATION

Not unlike the ‘let it crash’ philosophy of other highly scalable systems such as Erlang and Akka, Kubernetes expects that the applications it runs are designed to be resilient to restarts. That is, pods can crash, restart, or relocate (specifically, restart on a different node) at any time without warning, and application developers should design their applications to not rely on longevity of pods and their associated state. However, even when assuming that applications are designed to be restart resilient, there are many situations when it would be highly desirable, though not absolutely necessary, to retain state for as long as possible. For example, in the event of an in-memory cache service losing its state, the service will eventually repopulate its cache, but doing so takes time and can lead to less responsiveness for users of the affected system.

Live pod migration, with the characteristic of retaining most-to-all of pod state, is a valuable advantage over kill-and-restart whenever pods with local state (in-memory or within disk volumes) must be relocated. It can prove especially useful during time-sensitive mass relocations, because while infrequent restarts of a few pods within a sea of replicas is acceptable for most systems, losing the state of a large number of pods within a short time period may cause noticeable disruptions even in well-designed systems. Situations where live pod migration could be used for state-preserving pod relocation include:

- Performing node upgrades—either software or hardware upgrades could necessitate reboots with the former and cold swap of hardware components with the latter, and any pods running on affected node should be migrated away beforehand.
- Preparing for planned outages or reacting to partial outages—sudden and unplanned outages cannot be helped, but in the case of planned outages, or unplanned but partial outages (where ‘evacuation’ of pods is still possible), we can perform live migration on affected pods to minimize service disruption. Planned outages could happen either due to internal factors such as infrastructure upgrades, or external factors such as ISP maintenances.
- Dynamic switching of cloud providers—competing cloud infrastructure providers tend to have differences in pricing models such that during different times or levels of service load, the best choice of provider may change, and performing live migration to switch between them without causing service disruptions would be ideal. As a disclaimer, cross-cluster federation (especially between different cloud providers) within Kubernetes is still an active work in progress, so this specific use case of live migration may not be presently applicable.

In consideration of time constraints, our prototype for live pod migration aims only to demonstrate the successful use of Docker’s experimental checkpoint/restore operations within the context of Kubernetes, and by doing so be able to migrate a singular pod from one node to another. Our extensions provide Kubernetes users/administrators with the ability to initiate migrations by creating migration specs, and to monitor the progresses of migrations. These operations can be done either directly via the API server or through the command line tool known as *kubectl*. When migrating a pod, we ensure that the following crucial elements of a pod are migrated:

- Metadata from the original pod’s spec, such as labels used for pod querying and configuration options
- Local IP address of the pod to ensure correct restoration of network devices and sockets
- Disk volumes used by the pod’s containers
- All containers within the pod and their in-memory state

BREAKDOWN OF SECTIONS

The proceeding sections discuss our mechanism for live pod migration in more detail. They are organized as follows:

- System Design—a high level overview of the components involved and the procedures that they perform
- Implementation—a discussion of implementation details surrounding components that we created or extended
- Evaluation—how the mechanism performed under test scenarios and an analysis of cross-component communication
- Limitations—circumstances under which live pod migration fails and tradeoffs we made when designing our system
- Discussion—campfire stories about the struggles and discoveries that our team had while working on this project

System Design

Extending Kubernetes with live pod migration involves modifications and additions to three aspects of the existing system—modifications to the kubelet to allow for checkpointing and restoring of pods and containers, changes to the Kubernetes API and data representations, and creation of a new 'migration controller.' Many of our design choices are admittedly not ideal nor production-ready, but were made in order to build a demo-able prototype within tight time constraints.

CHECKPOINT AND RESTORE

Live migrating a pod involves migrating all of its individual containers. To do this, we take advantage of an experimental feature of Docker known as checkpoint and restore. In brief, a checkpoint operation will freeze a container and dump its in-memory state to a set of image files. The inverse operation restore will unfreeze a container and restore all of its in-memory state from a set of checkpoint image files. Nearly all container state is captured within these image files, with enough information to restore mounted disk volumes, open sockets, all allocated memory pages, and network configuration options such as IP address and port bindings. To migrate a container from node A to node B, we would perform the following steps:

1. Checkpoint the container on node A, dumping the image files into a known directory
2. Transfer the dumped image files from node A to node B
3. Create a new container (without running it) on node B, from the same container image as the original container
4. Restore the newly created container on node B using the image files it received from node A

Note that the newly created target container of restoration cannot be running when the restore operation is performed. One option is to simply create the container (which will pull the image if needed and register the container with Docker), but not run it. The second option is to create, run, then immediately checkpoint the container, such that it frozen. Both options would allow us to perform the restore operation, and in the latter option overriding any existing state that the container accumulated during its brief lifetime before being checkpointed and frozen.

API CHANGES AND ADDITIONS

In the Kubernetes architecture, all communication related to cluster state between users, controllers, and kubelets on nodes are done indirectly through API objects and the API server. Evidently, our modifications to the API include introducing additional pod specification flags, new pod statuses, and new migration-related object types and API endpoints.

Pod Specification Flags

The pod specification is the blueprint for a pod. It describes the containers within the pod, the pod's network configuration, and other options when running the pod. Whenever a pod specification of a running pod is changed, the change is propogated to the kubelet. In a sense, we can update the pod specification to indirectly trigger a change to the pod at will, such as checkpointing or restoring the pod. To do this, we add two boolean flags, *ShouldCheckpoint* and *ShouldRestore*, to the pod spec. The migration controller will set these flags when appropriate, and in response the kubelet will perform the sequence to checkpoint or restore all of the pods' containers, respectively.

Additionally, in the previous section on checkpoint and restore, we noted that a target container of restoration cannot be running when performing a restore. To set this up, we introduce a *DeferRun* flag to the pod spec, to be set at time of pod creation, which instructs the kubelet to create the pod and its containers, but immediately checkpoint and freeze them afterward. The newly created pod will then remain in checkpointed state until a restore occurs.

Pod Statuses

We introduce one new pod status to the system, *Checkpointed*. The *Checkpointed* status represents a pod in a condition where all of its constituent containers have been checkpointed. When this condition is met, a kubelet will update the pod status via the API server. In our design, a kubelet will properly detect and set the *Checkpointed* status, even if the kubelet itself was not the one who initiated the checkpoint. This is especially useful for manually testing our system's proper event handling, by invoking operations directly through Docker. A restored status is not necessary, since a restored container is simply back to *Running* state.

Migration Resource

Similarly to how users or controllers can direct the replication controller by creating replication spec objects, we add a new *Migration* resource, as well as associated RESTful "/migrations" endpoints and *kubectl* CLI commands. A *Migration* object consists of a *Migration Spec* and status information about the current progress of the migration. Apart from metadata such as name, labels, and timestamps, the *Migration Spec* contains only two effective fields—the first is the name of the pod to migrate, and the second is the name of the destination node to migrate to.

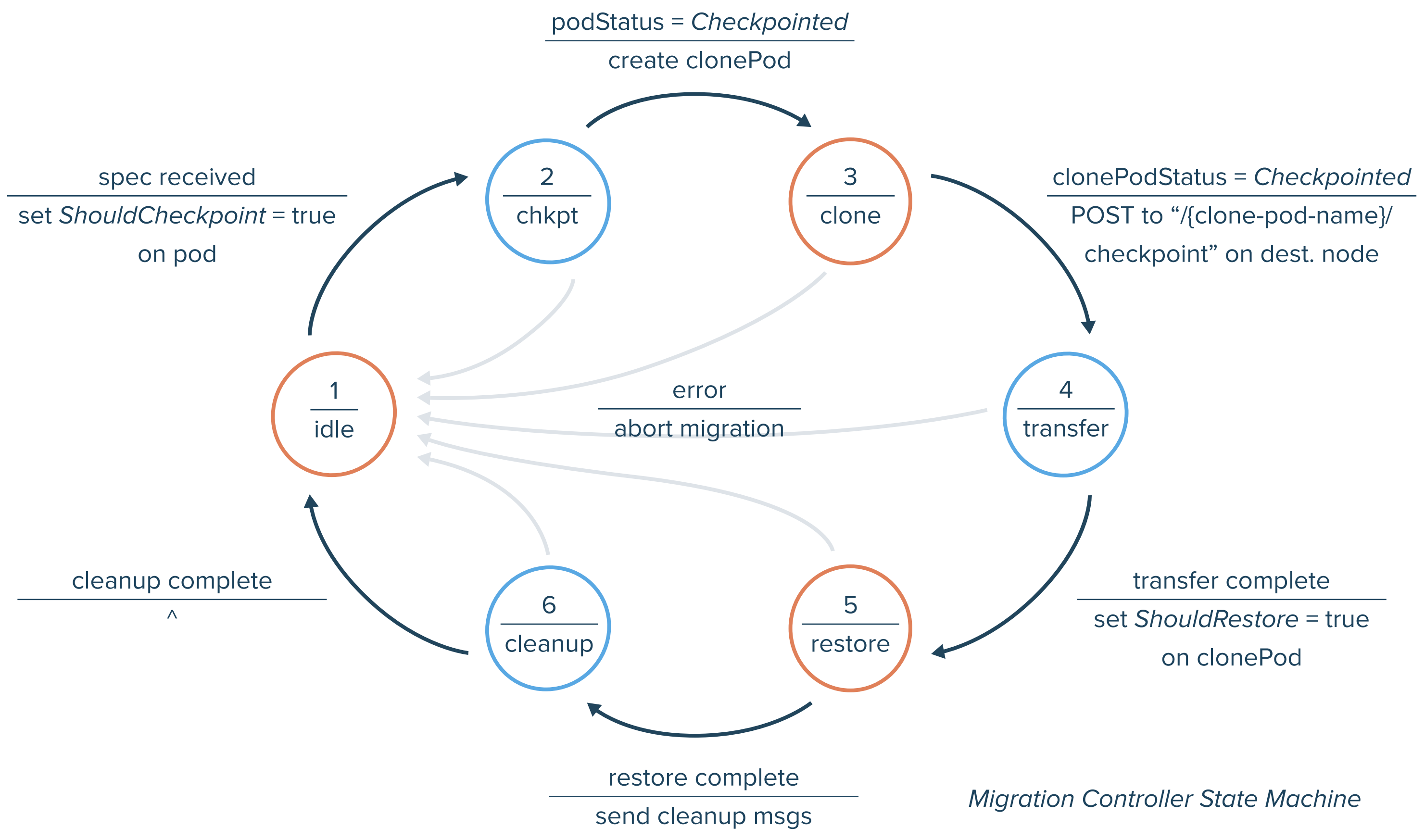
We extend *kubectl* such that the following commands can be made:

• Create a migration	`kubectl create -f <path-to-migration-spec-file>`
• List all active migrations	`kubectl get migrations`
• Get the json dump of a migration	`kubectl get migrations/<name-of-migration> -o json`
• Get details on a specific migration	`kubectl describe migration <name-of-migration>`

MIGRATION CONTROLLER

The migration controller is the component that coordinates migration efforts. It communicates with the participating kubelets primarily indirectly via the API server. Omitting minor implementation details around API calls, the migration controller roughly looks like this when represented as a state machine:

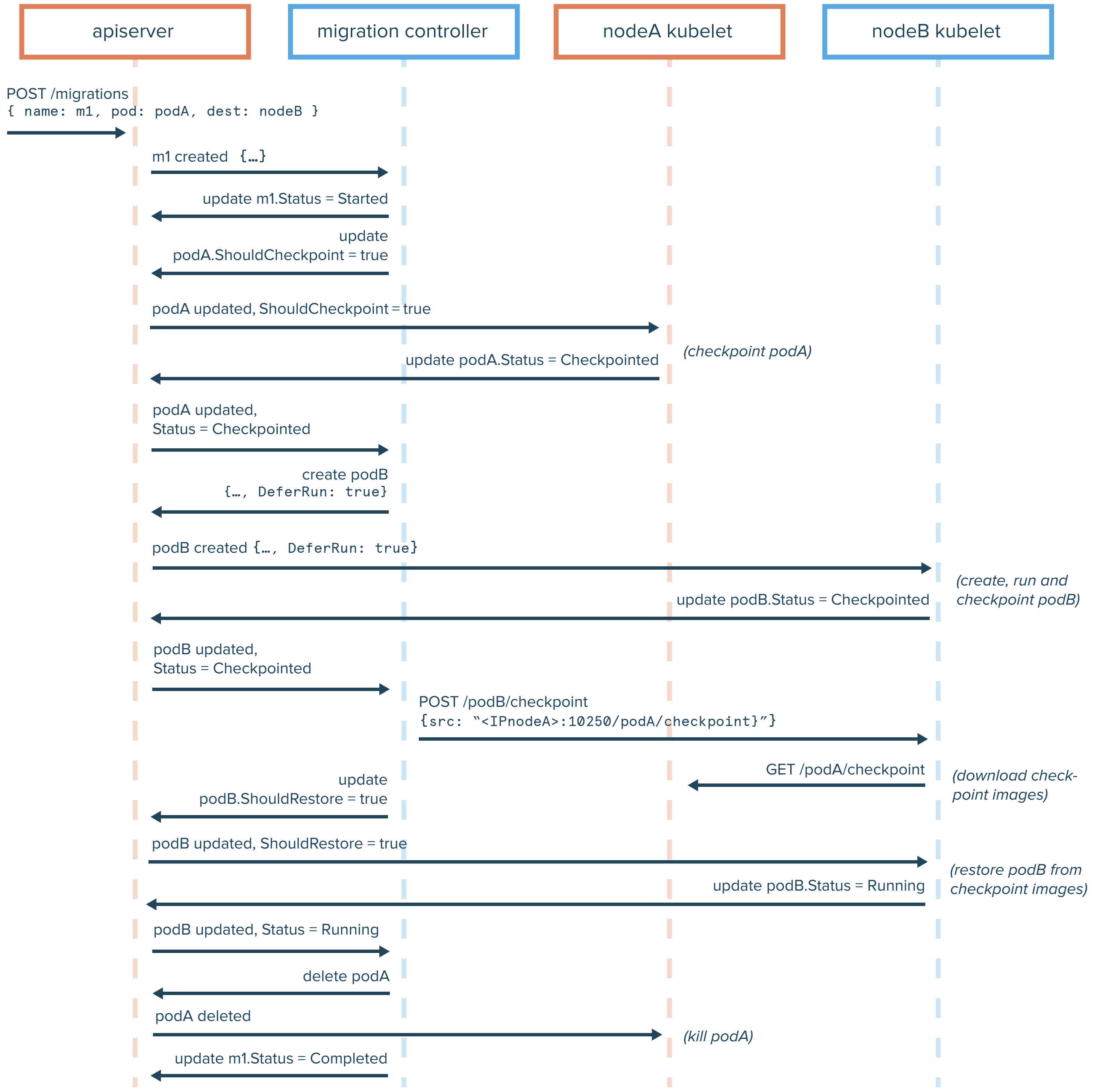
1. Idle—The controller is waiting to receive a migration spec.
2. Checkpointing—The controller has set the *ShouldCheckpoint* flag on the pod subject of migration. It awaits the pod status to be set to *Checkpointed* by the kubelet on the source node, which indicates that all containers of the pod have successfully been checkpointed.
3. Cloning—The controller has updated the API server with a new pod object that is a clone of the pod being migrated, except also with the *DeferRun* flag set. The pod and its containers will be created and immediately checkpointed and frozen. This pod is explicitly assigned to the destination node that was specified by the user in the *Migration Spec*. The controller is now awaiting the clone pod's status to be set to *Checkpointed* by the destination node's kubelet.
4. Transferring—The controller has sent an HTTP post request to the destination node's "/{clone-pod-name}/checkpoint" route with a body that contains the path on the source node from where it can download the checkpoint-dumped image files (in tar.gz format). The controller is waiting for the destination node to finish downloading the image files.
5. Restoring—The controller has set the *ShouldRestore* flag on the clone pod. It awaits the pod status to be set to Running by the destination node's kubelet, indicating successful state restoration.
6. Cleanup—The controller has finished the migration, and has sent cleanup messages to set the migration status to *Completed* and delete the source pod from the system. It is awaiting completion of these cleanup tasks, before resetting to the idle state.



Migration Controller State Machine

In practice, the migration controller can process multiple concurrent migrations on separate worker routines, as long as the pods being migrated are unique. The number of worker routines is by convention defaulted to 5.

The following diagram illustrates the communication sequence between components during a migration. Communications involving low level routines, the Docker daemon, and HTTP response messages have been simplified or omitted.


apiserver
migration controller
nodeA kubelet
nodeB kubelet
POST /migrations
{ name: m1, pod: podA, dest: nodeB }
m1 created {…}
update m1.Status = Started
update podA.ShouldCheckpoint = true
podA updated, ShouldCheckpoint = true
update podA.Status = Checkpointed
(checkpoint podA)
podA updated, Status = Checkpointed
create podB {…, DeferRun: true}
podB created {…, DeferRun: true}
(create, run and checkpoint podB)
update podB.Status = Checkpointed
podB updated, Status = Checkpointed
POST /podB/checkpoint
{src: "<IPnodeA>:10250/podA/checkpoint}"}
GET /podA/checkpoint
(download checkpoint images)
update podB.ShouldRestore = true
podB updated, ShouldRestore = true
(restore podB from checkpoint images)
update podB.Status = Running
podB updated, Status = Running
delete podA
podA deleted
(kill podA)
update m1.Status = Completed

DESIGN CONSTRAINTS

In an effort to reduce project scope, some design constraints were made:

- Node and network are expected to experience no failures during the period of migration, errors during migration will cause migration to abort without attempted cleanup
- Applications that communicate with other services via network must gracefully handle short bursts of service downtime during migration, or the migration may fail and simply restart the pod from scratch (though we have an incidental networking-related hurdle that we discuss in the limitations section)
- Pods, nodes, or migration objects that are actively part of migration should not be modified by any party other than the migration controller for the entire duration of the migration

Implementation

The implementation of our project involved forking and modifying three Go-based projects, namely Kubernetes, Docker, and go-dockerclient. Behind the scenes, we depend on the CRIU (Checkpoint Restore in Userspace) library for checkpointing and restoring linux processes, which in turn relies on relatively recent Linux kernel features. The net result is that our project has very specific requirements around operating systems and versions of binaries, which is not ideal nor convenient, but it's a lesson learned for playing with experimental software. In terms of dependencies, our development and testing environments look like this:

- Ubuntu 16.04 Xenial (Kernel-level checkpoint/restore enabled by default, easy access to CRIU 2.0 from launchpad)
- CRIU 2.0 (Docker-specific improvements over previous versions)
- Docker compiled from our fork (checkpoint/restore and static IP assignment features merged, minor bug fixes)
- go-dockerclient compiled from our fork (checkpoint/restore helper functions)
- Kubernetes compiled from our fork (live pod migration and static pod IP assignment)

DOCKER REQUIREMENTS AND MODIFICATIONS

The Docker checkpoint and restore features currently reside on a fork of Docker that is no longer actively developed. Checkpoint and restore is currently being re-implemented on top of Docker's newer container stack, including containerd and runC. However, given that the defunct checkpoint/restore fork of Docker was in fairly functional state, we decided to proceed with it for the purposes of our project.

Since the abandonment of the checkpoint/restore fork, a feature to allow static IP assignment of containers was developed and merged on the main Docker repo. We were able to merge this feature with the checkpoint/restore fork and build a custom binary, thus allowing us to migrate IP addresses of pods as part of our live migration process.

GO-DOCKERCLIENT MODIFICATIONS

Expectedly, due to the experimental nature of the checkpoint/restore feature, the Docker client library that Kubernetes uses did not have functions to invoke checkpoint/restore. We added the functions as needed, and updated Kubernetes to use our fork of the library.

KUBERNETES MODIFICATIONS

The bulk of our project work was done within Kubernetes itself, and major implementation areas correspond well to the aforementioned description of system design. Notable details of our implementation include:

- Creating an *etcd*-backed *Migration* object store—We referenced other object stores in the system, all with a very consistent pattern and set of boilerplate, and were able to build this fairly easily. Conceptually, the object store is very similar to typical RESTful web services and associated data access patterns. The store is essentially a client for the RESTful *etcd* API.
- Building the *Migration Controller*—Controllers in Kubernetes took some time researching and digging through source code for us to grasp. Its most confusing aspect at first glance is that it never communicates directly with any node, although its overreaching objective is to control what pods are running on nodes. Rather, it simply updates API objects and leaves the responsibility of noticing and fulfilling any changes to the kubelets themselves. Our migration controller, like other controllers, consists of a local cache of migration specs that watches and reflects changes from the API server, a work queue of migration specs to carry out, multiple worker routines that take ‘work’ from this queue, and a synchronization function that the worker routines run. The sychronization function in our case is a sequential procedure that performs one whole migration, or up until an error, before returning; it roughly corresponds to the state machine as described previously.
- Extending the *kubelet* with checkpoint and restore—Fundamentally, adding checkpoint and restore are simple calls to the Docker client library at the right times. However, due to the event-based reconciliation system of the *kubelet*, implementing checkpoint and restore necessitated treading carefully with container state and change events. Specifically, we had to update the event emitting and consuming components of the *kubelet* to not raise red flags upon checkpoint, which originally would initiate undesired pod restarts. Doing so also allowed us to create a non-running pod in preparation for a restore, whereas previously any pod with no running containers would be terminated and restarted automatically.
- Adding migration-related commands to *kubectl*—Since *kubectl* is essentially an HTTP client for the API server, and uses the same client abstraction libraries as the other components in the system, we were able to extend *kubectl* with little trouble.

Evaluation

Because of a networking issue that we faced, our evaluation of the system is split into results that test the effects of migration on networking, performed directly through Docker, and those that focus exclusively on cluster state and containers' memory state, performed via our live pod migration mechanism in Kubernetes. We recognize that non-network bound pods are not very realistic test cases, and we explain our challenges around this in the limitations section.

MEMORY-BACKED COUNTER

To assert the correct migration of in-memory state, we test migrating a pod with a single container through Kubernetes. The container contains a Go script that increments an internal counter every second and prints to stdout. We test this on both a local cluster setup (single node) and a multi-VM local cluster, though we quickly determined that due to the NAT-like networking that Docker provides, two containers on the same machine act roughly the same (as if they were two VMs) as when they are on two isolated virtual machines, barring external networking concerns around routing and iptables. After fixing a number of implementation issues, both test scenarios produce the expected behaviors.

We observe the outputs of the programs by hooking on to the outputs via the attach mechanism provided by Kubernetes. As expected, prior to migration, the counter prints out numbers up to its moment of checkpoint, for example 1–48. Post migration, the counter continues off from 49, with no smaller numbers in its output history. This primitive test also works correctly when performed directly using Docker's CLI.

With an image size of 6MB and a gigabit ethernet connection, image pulling time becomes negligible. In such an environment and on a local cluster setup, the entire migration process takes roughly 2–3 seconds.

REDIS-BACKED COUNTER

We extended the counter pod to contain two containers, one which is a Redis key-value store instance, and one which is a modified Go script that writes and reads its count from Redis before printing to stdout. In the event that it cannot connect to Redis, the counter app will continually sleep and retry. We replicated the previous test scenario, attempting on a single node (host machine) cluster and a multi-VM local cluster.

Due to the networking requirement (the counter container communicates with Redis via localhost addressing, so requires joining of network namespaces), this example 'migrates' successfully (i.e. the pods themselves do not crash) within Kubernetes but fails to restore network connectivity or proper functionality post-migration.

This time, the counter container is Ubuntu based (primarily to allow us to execute the bash binary and inspect network devices for debugging purposes), with a size of 68MB. The Redis container used is also Ubuntu based and is 72MB.

On the same local environment with gigabit ethernet as before, the initial test trial with multiple VMs took 11 seconds from the start to end of migration. This is primarily due to the repulling of the container images on the destination node. Repeated tests went down to roughly 5-7 seconds, after the machines have cached the container images locally. The additional time is to be expected given the increased image sizes (which have a small effect on launch time) and number of containers.

When this setup is repeated manually, by checkpointing via the Docker CLI, scp-ing the checkpoint files to another VM, and creating and restsoring the containers from that side, the restored counter produces the expected behavior of continuing to increment numbers from where it last left off. Note that we performed this operation in all possible orders of checkpointing and restoring–first checkpointing Redis, first checkpointing the counter container, first restoring Redis, and first restoring the counter container. The containers manage to migrate and continue running correctly in all cases, which we fully expect due to the counter app retrying whenever it cannot connect to Redis. We test our suspicions by modifying the counter script to crash immediately if it cannot connect to Redis, and as expected, when the Redis container is not running, the counter container exits immediately. Note that it is a characteristic of Docker to exit the container once its entrypoint program has finished executing.

To test external networking with metrics aside, we extend the counter container to also publish its count via an HTTP GET endpoint. This works exactly as expected when done through Docker (where each container owns its own network stack), and crashes miserably when done through Kubernetes (where all containers intricately join their network namespaces). In the latter case, the CRIU restore log mentions sockets failing to restore due to invalid network devices, but seeing as we aren't networking experts, we weren't sure what 'sockets' were. Nah, we're kidding, but in conclusion networking restoration works with Docker's network model, and not so much with Kubernetes'.

GENERATION OF SHIVIZ COMPATIBLE LOGS

Due to the intricate indirect and multilayered communication between components within Kubernetes, we were not able to use GoVector's feature of wrapping around network messages. Had we intercepted messages at a low level, our logs would contain huge amounts of noise from communication between components that are unrelated to our project. Indeed, Kubernetes outputs hundreds to thousands of lines of debug and event logs every minute. Our only feasible option, and the one we opted for, is to embed log and vector timestamp related data into the API objects that every one of our component reads from and writes to the central API server. Doing so allowed us to sequence events based on the order that these objects are accessed.

The three major parties involved in a migration, and the ones that we choose to log under, ignoring subroutines are:

- The migration controller
- The source node's kubelet
- The destination node's kubelet

Roughly speaking, the messages and events in our system that we choose to log include the following:

- Pod status updates written by *kubelets* and read by the migration controller as an indication of progress
- Creation and deletion of pods done by the migration controller
- HTTP requests and responses made directly between the controller and kubelets to transfer checkpoint image files
- Checkpoint and restore operations of individual containers for affected pods during a migration

Limitations

Considering that minimizing downtime is a major goal of live migration, our prototype design makes several sacrifices in that regard to keep the complexity reasonable. We also encountered a significant roadblock related to Kubernetes' networking model that we exhausted solution candidates for, but ultimately were unable to resolve by the project deadline.

POD FREEZING AND DOWNTIME

In our implementation of live pod migration, we freeze the subject pod prior to migrating it. In scenarios where container images (which are on the magnitude of hundreds of megabytes) are already cached on the destiantion node, the freeze-then-migrate procedure caused 1–5 seconds of downtime during our testing. However, just the time to pull an image from an online registry alone can extend the downtime to over 10 seconds in our current design. We initially attempted to mitigate this by pulling and creating the destination pod prior to checkpointing and freezing the source pod, but flipped this sequence when we implemented IP address migration. IP address migration requires the source pod to be frozen prior to creation of the destination pod in order to 'free' and transfer over the IP address. In a more ideal design, we would separate initializing of a pod into three independently invokable stages: pulling of images, creating of containers, and running/restoring those containers.

Another option is to leave the container running prior to checkpointing, however this necessitates multi-staged reconciliation of any change in state after the time of checkpoint. Performing this type of reconciliation is described in detail in papers around virtual machine live migration, but the procedure is relatively complex and not natively supported in Docker or CRIU in any way.

THE KUBERNETES NETWORK MODEL

The native Docker networking model by default assigns each container its own IP address and network namespace. Docker checkpoint and restore supports restoration of sockets and network devices gracefully, and we were able to migrate network-bound containers natively via the Docker CLI with no trouble.

Kubernetes' networking model deviates from that of Docker. As prior mentioned, IP addresses and network namespaces are pod scoped in Kubernetes. To achieve this, all containers within a pod 'join' their network namespaces with a special container that runs within every pod known as the pod infrastructure container. This special container 'owns' the networking for the pod, and becoming disjoint from this container means losing network connection irrecoverably. Indeed, whenever the infrastructure container fails within Kubernetes, all other containers in the pod must be restarted and rejoined to a new infrastructure container. To our knowledge, it is not possible to join a container's network namespace once it is created and running.

Using the Kubernetes networking model, our live pod migration system fails to migrate any containers with open network sockets due to this intricate weaving of network namespaces. This is a significant limitation that we may be able to work around only by bypassing the networking model entirely, and instead force Kubernetes to give every container within a pod its own IP address and networking stack. However, by doing so we essentially lose the pod abstraction of Kubernetes—clearly not an acceptable long term solution. Other possible solutions that we attempted include:

- Migrating pod IP addresses—This in and of itself works correctly, but it fails to address the namespace joining issue.
- Ensuring that applications listened on 0.0.0.0—We attempted this in order to prevent crashes during restoration whenever the network interface for a referenced IP address is not available. This partially solves the problem by preventing some socket-related crashes, but we still do not retain network connectivity post-migration.
- Migrating the infrastructure container—As a last resort, we hoped that making the infrastructure container itself a candidate of migration (in addition to ensuring it keeps the same IP address) would allow other restored containers to correctly rejoin their network stack. However, we could not successfully migrate the infrastructure container, instead resulting in cryptic errors in the CRIU debug log.

Though we were not able to confirm one way or another, it is entirely possible that the checkpoint/restore infrastructure that we depend on do not play well with joined network namespaces between containers.

Discussion

This shouldn't come as much of a surprise, but wow was working through this project a massive game of hide-and-seek every step of the way. The theme seems to be that whenever something is broken, usually it's because we did not properly update a component that we didn't even know existed. From initial research, it quickly became evident that most documentation around Kubernetes was centered around users of the system. There are a number of development related documents within the repository, but beyond a certain point the only way to understand the system fully is to dig and trace through source code.

In doing so, we learned quite a bit about development of large projects in Go, seeing techniques that we have not encountered previously. Among these include the vast set of code generation tools that Kubernetes includes, to generate things from conversions between API objects of different API versions (as part of a design that decouples API backward compatibility from core source code logic) and to generate static deepcopy functions (as opposed to reflection-based deepcopy). Godeps is used to manage external dependencies, and there are no *$GOPATH* struggles to be had, as it's all taken care of by the compilation shell scripts. Shell scripts exist that, when used as part of build scripts, ensure the purity of the codebase to some degree. For example, scripts exist to check proper gofmt-ing of all source files, proper inclusion of copyright statements and licenses, and proper inclusion of dependencies through Godeps.

Perhaps the most crucial development 'feature' that Kubernetes provided was the use of Google's glog library. The logs that glog records to files are are level-filterable, include exact source files and line numbers at which logs were printed, and capture any stack traces during errors. These should come as no surprise to developers who have worked at Google or any large production company. However, proper logging especially in a distributed system with a countless number of go-routines proved to be more valuable for debugging than printing to stdout (obviously), and using debuggers such as gdb.

Roles of Team Members

The breakdown of work among the team is roughly as enumerated below.

- *Andrea*—research and validation of Docker checkpoint/restore functionality, building demo-able Docker containers
- *Henry*—research on Kubernetes architecture, setup of Terraform for test environments and demo, pro diagram artist
- *Kelvin*—working with Docker networking and figuring out how to perform static IP address allocation, building demo-able Docker containers and coming up with our test cases, devising a method for outputting logs for ShiViz
- *Thomas*—research and development with Kubernetes and Docker

References

Verma, A., Pedrosa, L., Korupolu, M. R., Oppenheimer, D., Tune, E., & Wilkes, J. (2015). Large-scale cluster management at Google with Borg. Retrieved April 11, 2016, from *http://research.google.com/pubs/pub43438.html*

Clark, C., Fraser, K., Hand, S., Hansen, J. C., Jul, E., Limpach, C., Pratt, I., Warfield, A. (2005). Live Migration of Virtual Machines. Retrieved April 11, 2016, from *http://www.cl.cam.ac.uk/research/srg/netos/papers/2005-migration-nsdi-pre.pdf*