RANCHER

January 2018

Rancher 2.0: Technical Architecture

Version TP2

RANCHER

Contents

Background

We developed the Rancher container management platform to address the need to manage containers in production. Container technologies are developing quickly, and, as a result, the Rancher architecture continues to evolve.

When Rancher 1.0 shipped in early 2016, it included an easy-to-use container orchestration framework called Cattle. It also supported a variety of industry-standard container orchestrators, including Swarm, Mesos, and Kubernetes. Early Rancher users loved the idea of adopting a management platform that gave them the choice of container orchestration frameworks.

In the last year, however, the growth of Kubernetes has far outpaced other orchestrators. Rancher users are increasingly demanding better user experience and more functionality on top of Kubernetes. We have therefore decided to re-architect Rancher 2.0 to focus solely on Kubernetes technology.

Kubernetes is Everywhere

When we started to build Kubernetes support into Rancher in 2015, the biggest challenge we faced was how to install and set up Kubernetes clusters. Off-the-shelf Kubernetes scripts and tools were difficult to use and unreliable. Rancher made it easy to set up a Kubernetes cluster with a click of a button. Better yet, Rancher enabled you to set up Kubernetes clusters on any infrastructure, including public cloud, vSphere clusters, and bare metal servers. As a result, Rancher quickly became one of the most popular ways to launch Kubernetes clusters.

In early 2016, numerous off-the-shelf and third-party installers for Kubernetes became available. The challenge was no longer how to install and configure Kubernetes, but how to operate and upgrade Kubernetes clusters on an on-going basis. Rancher made it easy to operate and upgrade Kubernetes clusters and its associated etcd database.

By the end of 2016, we started to notice that the value of Kubernetes operations software was rapidly diminishing. Two factors contributed to this trend. First, open-source tools, such as Kubernetes Operations (kops), have reached a level of maturity that made it easy for many organizations to operate Kubernetes on AWS. Second, Kubernetes-as-a-service started to gain popularity. A Google Cloud user, for example, no longer needed to set up and operate their own clusters. They could use Google Container Engine (GKE) instead.

The popularity of Kubernetes continues to grow in 2017. The momentum is not slowing. Amazon Web Services (AWS) announced Elastic Container Service for Kubernetes (EKS) in November 2017. Kubernetes-as-a-service is available from all major cloud providers. Unless they use VMware clusters and bare metal servers, DevOps teams will no longer need to operate Kubernetes clusters themselves. The only remaining challenge will be how to manage and utilize Kubernetes clusters, which are available everywhere.

Rancher 2.0: Built on Kubernetes

Rancher 2.0 is a complete container management platform built on Kubernetes. As illustrated in Figure 1, Rancher 2.0 contains three major components.

Rancher Kubernetes Engine (RKE)

RKE is an extremely simple, lightning fast Kubernetes installer that works everywhere. RKE is particularly useful in standing up Kubernetes clusters on VMware clusters, bare metal servers, and VM instances on clouds that do not yet support Kubernetes service.

RKE can set up a Kubernetes cluster on a given set of nodes. RKE can additionally add nodes to or remove nodes from the cluster.

Unified Cluster Management

Rancher 2.0 integrates with managed Kubernetes services such as GKE, EKS, and AKS. Rancher 2.0 invokes RKE to provision and operate Kubernetes clusters when managed Kubernetes services are not available.

Rancher 2.0 provides an authentication proxy for all Kubernetes clusters under management. This enables, for example, an enterprise developer to sign into a GKE cluster using the Active Directory credentials managed by corporate IT.

Building on the native RBAC capabilities of Kubernetes, Rancher 2.0 enables IT administrators to configure and enforce access control and security policies across multiple Kubernetes clusters.

Rancher 2.0 provides visibility into capacity and cost of underlying resources consumed by Kubernetes clusters.

Application Workload Management

Rancher 2.0 offers an intuitive user interface for managing application workloads. Users can create containers, services, DNS, load balancers, and applications without learning Kubernetes terminologies. Much of the Cattle experience in Rancher 1.0 is available and is built on the powerful Kubernetes orchestration engine.

The app catalog in Rancher 1.0 has been expanded to support both compose templates and Helm templates. Support for additional catalog templates can be added in the future.

Rancher 2.0 works with any CI/CD systems that integrate with Kubernetes. For example, Jenkins, Drone, and GitLab will continue to work with Rancher 2.0 as they did with Rancher 1.0. Rancher 2.0 will additionally include a managed CI/CD service built on Jenkins. The Rancher 2.0 CI/CD service can be deployed from the catalog and will seamlessly integrate with the Rancher UI.

Rancher 2.0 works with any monitoring and logging systems that support Kubernetes. For example, DataDog, Sysdig, and ELK will continue to work with Rancher 2.0 as they did with Rancher 1.0. Rancher 2.0 will additionally include a managed Fluentd, ElasticSearch, and Kibana service for log aggregation. Rancher 2.0 will also include a managed Prometheus service for out-of-the-box monitoring.

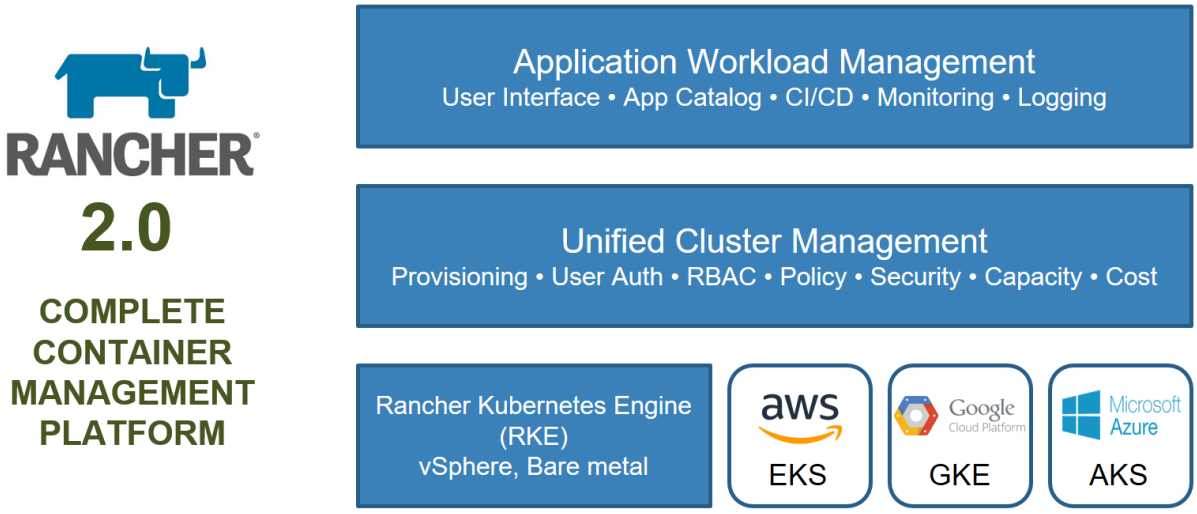

Figure 1 Overview of Rancher 2.0

High-level Architecture

Like Rancher 1.0, the majority of Rancher 2.0 software runs on the Rancher server. Rancher server includes all the software components used to manage the entire Rancher deployment.

Figure 2 illustrates the high-level architecture of Rancher 2.0. The figure depicts a Rancher server installation that manages two Kubernetes clusters: one Kubernetes cluster created by RKE and another Kubernetes cluster created by GKE.

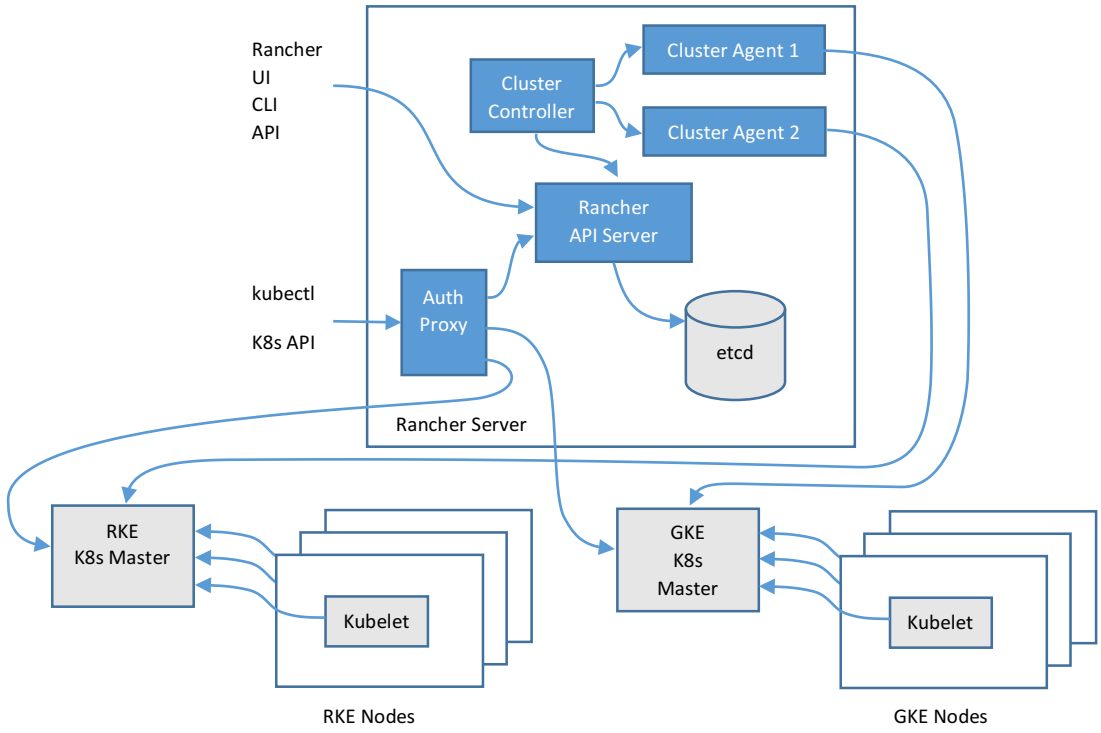

Figure 2 Rancher 2.0 High Level Architecture

Rancher Server Components

In this section we describe the functionality of each Rancher server component.

Rancher API Server

Rancher API server is built on top of an embedded Kubernetes API server and etcd database. It implements the following functionality:

1. User management. Rancher API server manages user identities that correspond to external authentication providers like Active Directory or GitHub.
2. Authorization. Rancher API server manages access control and security policies.
3. Projects. A project is a grouping of multiple namespaces within a cluster.
4. Nodes. Rancher API server tracks identities of all the nodes in all clusters.

Cluster Controller and Agents

The cluster controller and cluster agents implement the business logic required to manage Kubernetes clusters. All the logic that is global to the entire Rancher install is implemented by the cluster controller. A separate cluster agent instance implements the logic required for the corresponding cluster.

Cluster agents perform the following activities:

1. Manage workload. This includes, for example, creating pods and deployments in each cluster.
2. Applying roles and bindings that are defined in global policies into every cluster.

3. Propagate information from cluster to rancher server: events, stats, node info, and health.

The cluster controller performs the following activities:

1. Configures access control policies to clusters and projects.
2. Provisions clusters by invoking the necessary Docker machine drivers and invoking Kubernetes engines like RKE and GKE.

Authentication Proxy

The authentication proxy proxies all Kubernetes API calls. It integrates with authentication services like local authentication, Active Directory, and GitHub. On every Kubernetes API call, the authentication proxy authenticates the caller and sets the proper Kubernetes impersonation headers before forwarding the call to Kubernetes masters. Rancher communicates with Kubernetes clusters using a service account.

Rancher API Server

Rancher API server is built on top of an embedded Kubernetes API server and etcd database. All Rancher specific resources that are being created using Rancher API get translated to CRD (Custom Resource Definition) objects with their lifecycle being managed by one or several Rancher controllers.

When it comes to Kubernetes resources management, there are several code patterns followed by Kubernetes open source community developing controllers in Go programming language, most of them involving use of the client-go library (https://github.com/kubernetes/client-go). The library has nice utilities like Informer/SharedInformer/TaskQueue making it easy to watch and react on resource changes as well as maintaining in-memory cache to minimize number of direct calls to the API server. The Rancher API framework **extends** client-go functionality to save the user from writing custom code for managing generic things like finalizers and condition updates for the objects by introducing Object Lifecycle and Conditions management frameworks; adds better abstraction for SharedInformer/TaskQueue bundle using GenericController.

For the User facing APIs, the Rancher API framework adds features like sorting, object links filtering fields based on user permissions, pluggable validators and formatters for CRD fields.

To summarize the above, the Rancher API framework provides:

- User facing API schema generation with an ability to plug custom formatters and validators.
- Controller interfaces generation for CRDs and native Kubernetes objects types
- Object lifecycle management framework
- Conditions management framework
- Simplified generic controller implementation by encapsulating TaskQueue and SharedInformer logic into a single interface

Appendix A shows an example of how the Rancher API server is used to implement a Rancher API.

Authentication and Authorization

Rancher delivers a robust and user-friendly authentication and authorization system by building on the solid foundation of authentication and authorization primitives already in Kubernetes.

At a high level, Rancher enhances and improves Kubernetes authentication and authorization with the following functionality:

- User and group management as first class functionality
- Grouping of namespaces into projects
- Robust role management
- Integration with multiple authentication providers regardless of how or where your Kubernetes control plane is deployed using an authentication proxy
- Cross-cluster management of projects, users, and groups from a single pane of glass

To put that in perspective, here is a brief explanation of the authentication and authorization primitives provided by Kubernetes:

- For user and group management, users and groups do not exist as a first-class concept in Kubernetes. Instead, user identities are treated as opaque strings when setting and evaluating permissions. One practical impact of this is that maintaining user and group permissions across multiple namespaces and clusters can be quite arduous.
- For grouping resources, Kubernetes provides two scopes: global and namespace. This fails to meet scenarios where groups of users should have the same permissions across multiple namespaces leads to either labor intensive role management or to very coarse-grained namespaces that contain too many applications.
- For role management, Kubernetes provides the ClusterRole (defined at the global scope) and Role (defined at the namespace scope) resources. These generally cover the needs of single-cluster management but when leveraging more than one cluster, the burden is on the user to maintain ClusterRoles and Roles across clusters
- For integration with authentication providers, Kubernetes has an excellent plugin mechanism that allows teams deploying their own clusters to integrate with many providers. However, this feature is unavailable to those wishing to leverage cloud-based Kubernetes offerings such as GKE.
- Cross-cluster management of resources is still in its nascent stage in native Kubernetes and the long-term direction the technology will take is still unknown.

The following sections will describe each of above Rancher features in greater detail.

User and group management

Through the Rancher API and UI, users can be created, viewed, updated, and deleted. Users can be associated with external authentication provider identities (like GitHub or LDAP) and a local username and password. This provides greater flexibility and guards against downtime at the authentication provider. Similarly, groups can be created, viewed, updated and removed from the Rancher API and UI. A group membership can be a mix of explicitly defined users or group identities from the authentication provider. For example, you can map easily map a LDAP group to a Rancher group.

Projects

Using the Rancher API or UI, users can create projects and assign users or groups to them. Users can be assigned different roles within a project and across multiple projects. For example, a developer can be given full create/read/update/delete privilege in a "dev" project but just read-only access in the staging

and production projects. Furthermore, with Rancher namespaces are scoped to projects. So, users can only create, modify, or delete namespaces in the projects they are a member of. This greatly enhances the multi-tenant self-service functionality of Kubernetes.

Role management

The primary value that Rancher brings to the table with role management is the ability to manage these roles (and to whom they are assigned) from a single global interface that spans clusters, projects, and namespaces. To achieve this, Rancher has introduced the concept of "role templates" that are synchronized across all clusters.

Authentication provider integration

As stated previously, while Kubernetes has a good authentication provider plugin framework, that functionality simply isn't available when using Kubernetes from a cloud provider. To overcome this, Rancher provides an authentication proxy that sits in front of Kubernetes. This means that regardless of where or how your clusters are deployed, the same authentication provider can be used.

Cross-cluster management

Kubernetes is largely focus on single-cluster deployments at this point. In contrast, Rancher assumes multi-cluster deployments from the start. As a result, all authentication and authorization features are designed to span large multi-cluster deployments. This greatly simplifies the burden of onboarding and managing new users and teams.

Upgrade

Rancher 1.0 was built on Docker, and cannot be upgraded to a Kubernetes cluster without workload disruption. Rancher 1.0 users must setup a separate Rancher 2.0 cluster, migrate the workloads, and decommission the Rancher 1.0 cluster.

Users can upgrade to new versions Rancher 2.0 by upgrading the Rancher server. Rancher 2.0 handles RKE cluster upgrades. Rancher 2.0 integrates with cloud providers like GKE to upgrade GKE clusters. Rancher 2.0 does not attempt to upgrade imported Kubernetes clusters.

High Availability

By building the Rancher server on the Kubernetes API server and etcd database, the Rancher server is highly available as long as there are multiple instances of Kubernetes API server and there is an HA etcd cluster.

A typical HA Rancher deployment, for example, may consist of 3 nodes, each running one instance of the API server and the etcd database.

Scalability

Scalability of Kubernetes Clusters

As of Kubernetes version 1.6, A Kubernetes cluster can scale to 5,000 nodes and 150,000 pods. User can expect Rancher 2.0 to manage clusters up to that scale as well.

Scalability of Rancher Server

There is no inherent limit on how many Kubernetes clusters each Rancher server can manage. We do not expect an issue for Rancher 2.0 to manage up to 1,000 clusters.

The real scalability limits of Rancher server are:

1. Total nodes across all clusters
2. Users and groups
3. Events collected from all clusters

Rancher server stores all the above entities in its own database. We will improve scalability along these dimensions over time to meet user needs.

Appendix A: How Rancher API v3/Cluster objects are implemented

If you are a developer, and need to add new object and functionality for Kubernetes using the Rancher API framework, the sections below would explain the steps. We are going to use Rancher v3/Cluster object as an example.

Step 1. Define the object and generate the schema/controller interfaces

Any Rancher CRD object has to be first defined in the **Types** project (https://github.com/rancher/types) under the corresponding api group (types/apis/<group>/<version>).

In order to expose object via Rancher APIs, it also has to be defined in types/apis/<group>/<version>/schema/schema.go. The schema file object can also be customized to hide or rename a particular field to make it more user friendly.

Examples for v3/Cluster type:

- Object definition: https://github.com/rancher/types/blob/master/apis/management.cattle.io/v3/cluster_types.go
- Schema definition: https://github.com/rancher/types/blob/master/apis/management.cattle.io/v3/schema/schema.go

Once the object is defined**, Types** will generate the schema/controller. This auto generated code will be used by the Rancher API server and custom controllers.

Example for v3/Cluster type:

- Controller interface: https://github.com/rancher/types/blob/master/apis/management.cattle.io/v3/zz_generated_cluster_controller.go
- Schema interface: https://github.com/rancher/types/blob/master/client/management/v3/zz_generated_cluster.go

Step 2: Add custom logic to API validation

Management-api project (https://github.com/rancher/management-api provides user-facing APIs based on schemas defined in Types and defines custom logic for API actions/formatters/validators. If you need to run the check for any field of your object, or hide certain operation on the object based on the object's state, that would the place to do it.

Examples of formatters/validators can be found here: https://github.com/rancher/management-api/tree/master/api

Step 3: Define object management logic using custom controller

Any object in Kubernetes goes over Create/Update/Remove processes. If you are to add any custom logic behind these operations, you'd need to write a custom controller. The Rancher API framework defines a set of methods helping user to manage Kubernetes resource in easy and elegant way. First thing you need to do is create a Kubernetes client using github.com/rancher/types/config package; example for v3/Cluster controller can be found here: https://github.com/rancher/cluster-controller/blob/master/main.go . The client will give you an access to the CRD and core objects schemas, controllers generated on step 1, and provide the shared cache.

3.1 Object lifecycle management

Rancher API Framework lifecycle management interface lets custom controller to properly initialize the object fields during the Create with a minimum chance for conflicts; provides automatic injection for Finalizer in order for the controller to execute a cleanup before the object is gone from Kubernetes API and underlying cache on Remove. Example of use for v3/Cluster provisioner can be found here: https://github.com/rancher/cluster-controller/blob/master/controller/clusterprovisioner/provisioner.go

3.2 Generic controller

If object lifecycle management is not required, but there is still a need to listen and react on object's events (example: v3/Cluster stats aggregation based on Node stats changes), Rancher API framework provides a more lightweight GenericController - a wrapper around client-go TaskQueue and SharedInformer, with simplified interface. All you have to do is, register the handler for the events you are interested in, and define the callback method. No need to initialize the taskQueue or define a shared informer; the framework will do it for you automatically. Example for cluster stats aggregation using sync can be found here: https://github.com/rancher/cluster-controller/blob/master/controller/clusterstats/statsaggregator.go

3.3 Object conditions management

The Rancher API Conditions management framework lets you define the condition that has to be updated on the underlying CRD object based on the implementation outcome. Example: condition "Provisioned" will be set to true on the cluster when the provision completed successfully; and false – with the reason and message for the failure – if the operation failed. The only thing that you have to pass to the framework in this case is the callback method that drives the condition reset; there is no need to update the Condition field inside the controller – the framework will do it for you. Example of conditions reset for v3/Cluster based on provisioning state, can be found here: https://github.com/rancher/cluster-controller/blob/master/controller/clusterprovisioner/provisioner.go

More examples of Rancher custom controllers can be found under following projects:

https://github.com/rancher/cluster-controller

https://github.com/rancher/cluster-agent

https://github.com/rancher/workload-controller