Google Summer of Code

PROPOSAL

Storage API for Aggregated API Servers

By Marko Mudrinić

Author:
Marko Mudrinić,
Faculty of Mathematics,
University of Belgrade

Table of Contents

Abstract

Kubernetes offers two ways to extend the core API, by using the CustomResourceDefinitons or by setting up an aggregated API server. This ensures users don't need to modify the core API in order to add the features needed for their workflow, which later ensures the more stable and secure core API.

One missing part is how to efficiently store data used by aggregated API servers. This project implements a Storage API, with a main goal to share the cluster's main etcd server with the Aggregated API Servers, allowing it to use cluster's main etcd just like it would use it's own etcd server.

Background

Aggregated API servers are API servers that sit behind the kube-apiserver, which acts as a proxy to the aggregated one. To users, the aggregated API server appears like the core Kubernetes API is extended.

This is a very powerful mechanism, allowing users to easily customize the API per their need, but without need to modify the core Kubernetes API, ensuring it remains stable and secure.

The aggregated API servers have a possibility to use a custom storage backend. For example, if you're extending Kubernetes API with the metrics API server, you can deploy and use the time-series storage backend, which is better suited for that use case compared to the key-value solutions such as etcd.

However, it's not always suitable and practical to deploy your own storage backend when you want to use aggregated API servers. Kubernetes uses etcd, and often you want to use the same etcd cluster as used by Kubernetes for your aggregated API server. This removes the overhead from maintaining another etcd cluster, saves the cluster's resources and makes it easier to backup data, as you don't need to separate backup strategies.

The problem which arises here is how to securely allow access to the cluster's main etcd. Currently, to access the cluster's etcd you would need to obtain certificates and network access to the etcd cluster. Even if you obtain the prerequisites, that would give the aggregated API server full access to data in the cluster's main etcd, which represents a big security problem. Another problem is that sometimes etcd is hidden from the aggregated API server operator, for example Google Container Engine doesn't give you access to etcd.

This project creates and exposes an API in order to allow access to the cluster's main etcd and solves above mentioned problems. The end goal of the project is to simplify operations for operators, by allowing them to reuse the cluster's main etcd. We also want to let operators to

use standalone etcd servers with this API if they want to do so, as well as to ensure that third-party aggregated API servers can easily be installed using tools such as “helm”.

References

If you want to learn more about aggregated API servers and how they compare to CustomResourceDefinitions, I recommend the following resources:

- Extending the Kubernetes API with the aggregation layer.
- Comparison between CustomResourceDefinitions and aggregated API servers.
- “Extending Kubernetes: Our Journey & Roadmap” talk by Daniel Smith and Eric Tune.

Project Implementation

This part of the proposal explains how I plan to implement the Storage API.

Goals

In this project I want to create an API to be used by the aggregated API servers, in order to access the cluster's main etcd, just like they would access their own etcd cluster.

Beside using the API to access the cluster's main etcd, we want to allow cluster operators to use standalone etcd clusters. For example, if cluster's main etcd is already under high load, it's not a good idea to let an aggregated API server use it as well.

This API can also be used by cloud providers to offer standalone etcd instances as a service, to be used by third-party aggregated API servers.

Because of this, we want to ensure users can use this API with standalone etcd instances like they would use it with cluster's main etcd.

One of the main goals of the project is to integrate the Storage API into kube-apiserver. This ensures easier communication between the aggregated API servers and the Storage API. However, the Storage API is supposed to be able to work in standalone mode as well, so if we decide not to integrate it into kube-apiserver, it can still work outside the kube-apiserver without modifications.

The end goal is to let users to install third-party aggregated API servers by leveraging the Storage API and tools such as "helm".

Non-Goals

Beside goals described above, we'll consider allowing operators to throttle the access to the etcd and to put quotas on aggregated API server's namespace.

By letting operators to throttle access from the aggregated API servers to the etcd, we could prioritize traffic, for example if users want to prioritize kube-apiserver etcd requests, or just to prioritize specific aggregated API server.

By placing quotas, operators could make sure that the aggregated API server will never use more space and resources than allowed, so kube-apiserver (or other aggregated API server) can always work as expected.

Reusing the cluster's main etcd

The most efficient way to expose a storage backend for the aggregated API servers is to reuse the cluster's main etcd. This removes the overhead of deploying and maintaining the dedicated etcd cluster, maintaining backups, as well as saves cluster resources and operators time.

If we would just allow aggregated API servers to read and write to the cluster's etcd, we would need to totally trust the aggregated API server, which is not always possible, especially for third-party API servers. By exposing an API we can control what etcd data aggregated API servers can access.

Implementing the Storage API

In order to perform CRUD operations against etcd, aggregated API servers are supposed to talk to the Storage API.

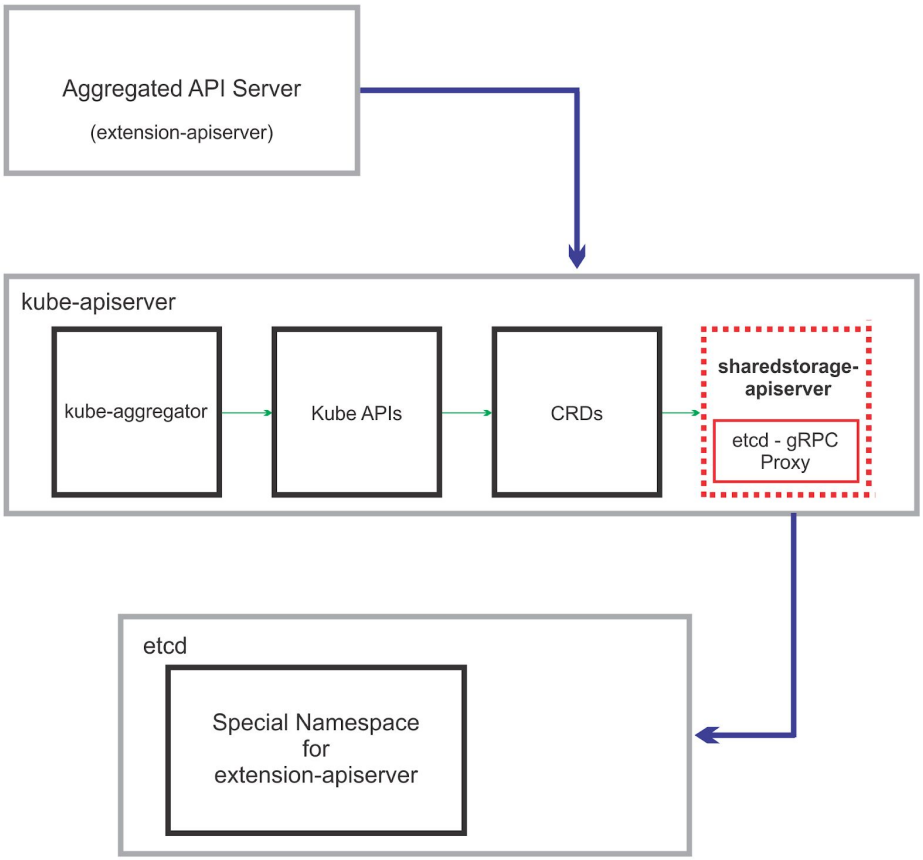
The main goal is to integrate the Storage API into the kube-apiserver, as in that case the Storage API server component receives the network access to etcd and the needed certificates, on the activation time via delegation.

The requests flow in that case is: the aggregated API server sends a request to the kube-apiserver, where kube-aggregator receives it. Then, the kube-aggregator sends that request through the delegation chain to the first delegated API server in the chain.

Delegated API servers are actively listening for incoming requests. If the request isn't supposed for that API server, the API server sends it to the next delegated API server in the chain.

Once the Storage API receives its request, it starts the etcd-gRPC proxy against the requested namespace and then performs the requested operations.

The following schema shows how communication between components works in case when the Storage API is integrated into the kube-apiserver. The "sharedstorage-apiserver" represents the Storage API. Green arrows inside the kube-apiserver represents the delegation chain.



Running Storage API in Standalone Mode

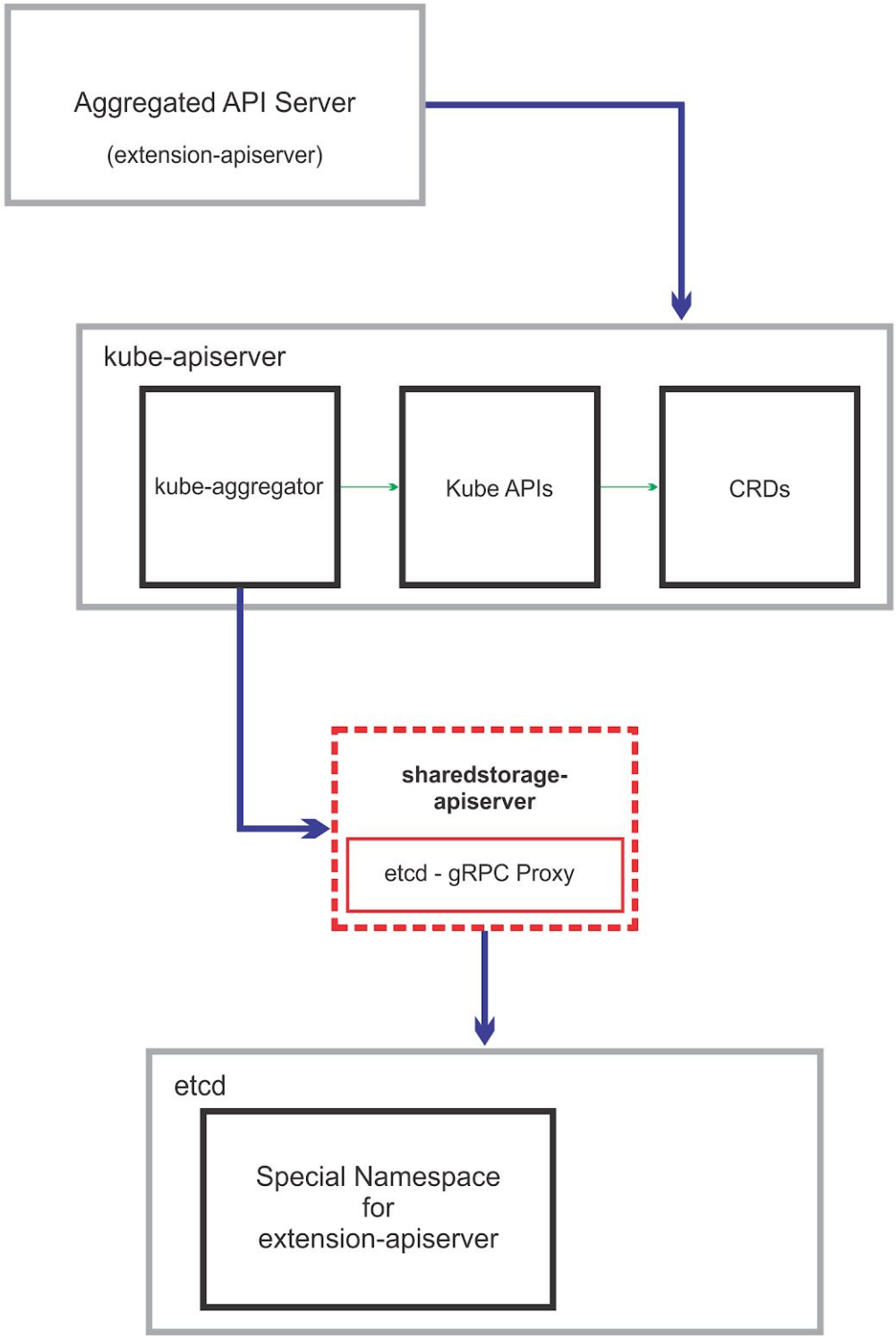
If we decide not to integrate the Storage API into the kube-apiserver, we want to make sure it's able to work in the standalone mode without modifications.

In this case the Storage API is not compiled into the kube-apiserver, instead it's running as a standalone component between the kube-apiserver and etcd. Unlike when it's running as part of the kube-apiserver, we need to ensure the Storage API has the network access to etcd and needed certificates to securely access it.

The aggregated API servers are sending requests to the kube-apiserver, where kube-aggregator is proxying those requests to the Storage API, which is running outside the kube-apiserver.

Once request is received by the Storage API, the workflow is the same as when the API is integrated in the kube-apiserver: the Storage API creates a etcd-gRPC Proxy against the requested namespace, and then performs the requested operations.

The following schema shows how aggregated API servers are communicating with the Storage API in this case. Same as for the previous schema, the “sharedstorage-apiserver” represents the API I’m implementing in this project.



Communication with API and etcd

To ensure secure communication between the Storage API and etcd, we're going to use the etcd's gRPC Proxy. The plan is to vendor etcd into the the Storage API code, so we don't need special component outside the Storage API for running the gRPC Proxy. This ensures the Storage API is observed as a single component by the cluster operators.

The gRPC proxy proxies all requests from the Storage API to the dedicated namespace. The aggregated API server can only access its own namespace using the Storage API.

The Storage API will have its own API group, let's call it "sharedstorage.k8s.io". In other words, the Storage API will be reachable via endpoints under "/apis/sharedstorage.k8s.io". The etcd-gRPC proxy will be provided as a Storage API's subresource under a resource "etcdnamespace" (names might change).

In that case, we'll have "/apis/sharedstorage.k8s.io/etcdnamespace/v1alpha1" for creating and managing etcd namespaces. Each aggregated API server will get an endpoint, such as: "/apis/sharedstorage.k8s.io/etcdnamespace/v1alpha1/sample-apiserver/etcd", i.e. with the subresource "/etcd" providing the etcd-gRPC proxy. Via that endpoint, aggregated API servers are able to modify data only in their own namespace.

It will be up to discussion whether the "etcdnamespace" resource will be cluster-wide or namespaced.

This approach improves security and ease of usability by the large margin, as we can't directly access all cluster's data, and we don't need to set up complex networking infrastructure between aggregated API servers and etcd.

Using Namespaces

One of the biggest security risks we have to solve is to ensure that the aggregated API servers can't access etcd keys that they don't own.

Namespaces allow us to isolate a subset of keys, under the specific prefix, let's say "/prefix". All aggregated API server's data is going to be stored in that prefix.

The etcd-gRPC proxy is used to start etcd server only with access to that prefix. All keys under that prefix are observed as roots keys from the etcd-gRPC proxy for that namespace. That ensures aggregated API servers can't access outside the root key, or in other words, they can't access the data they don't own.

For etcd, all keys for the Aggregated API server are located under a “/prefix” prefix (how this prefix looks like exactly and whether the etcdnamespace name maps directly to a prefix is to be discussed).

Namespaces requires etcd version 3.2+, however Kubernetes already uses etcd 3.2.16, which supports namespacing, so etcd upgrade is not needed.

Certificates

In order to access the etcd cluster, you need to obtain certificates from the cluster/cluster operator.

In the case when the Storage API is compiled in the kube-apiserver, the API already has access to certificates, so it’s not need to supply them explicitly.

To make the Storage API works in the standalone mode, we need to implement the set of command-line flags for providing etcd server URLs and certificates.

To ensure that Storage API can work flawlessly in both deployment scenarios (integrated in kube-apiserver or running in the standalone mode), it’s required to fix the issue kubernetes/kubernetes#60582.

Prototype

In order to show how is this supposed to work, I’ve created simple etcd-gRPC Proxy prototype that you can find on xmudrii/etcdproxy-proof-of-concept.

To run the prototype download the release from GitHub Releases or clone the repository and use “go install” to install the CLI. You can run the API server by executing the “etcdproxy-proof-of-concept start” command. To specify the namespace, use the “--namespace” flag. For other flags, such as if you want to change etcd address, check use the “--help” flag.

By the time of writing the proposal, only write operation is supported. The plan is to implement other operations as well, so check the README page for updates.

To write to a key, you need to send a HTTP “PUT” request to the “/write” endpoint, with JSON payload contained of “name” and “value” keys. By default, the API server is running on the port “:8000”, but you can change it using the “ETCDAPISERVER_PORT” environment variable. Example command:
“curl -X PUT -d '{"name": "version", "value": "v1.9.2"}' http://127.0.0.1:8000/write”

Then you can verify that key is correctly written using “etcdctl”. Let’s say that main etcd is running on the port “:2379” and proxied one is running on the port “:23790”, and that proxy is operating on the “k8s” namespace.

The following command returns the key’s value: “v1.9.2”:
“ETCDCTL_API=3 etcdctl --endpoints=http://localhost:23790 get version”

However, if you try to get “k8s/version” or any other value that’s not in the namespace, nothing will be returned.

To check that the key is written to the main etcd, you can use the following command:
“ETCDCTL_API=3 etcdctl --endpoints=http://localhost:23790 get k8s/version”

Similar as before, if you try to get just “version”, nothing will be returned.

References

The following issues and documents contain the important information about the Storage API:

- Issue #46351 — Decide on an approach for shared aggregated API server storage.
- Storage for Extension API Servers document by @lavalamp and SIG API-Machinery.
- etcd-gPRC Proxy documentation.
- etcd-gRPC Proxy Prototype.
- Issue #60582 — Defaulting mechanism for APIService.caBundle missing.
- Three parts blog series by Stefan Schimanski and Michael Hausenblas:
  Kubernetes deep dive: API Server - part 1,
  Kubernetes Deep Dive: API Server - part 2,
  Kubernetes Deep Dive: API Server - part 3a.

Schedule of Deliverables

There are four milestones for successfully completing this project:

1. Proposal draft for sharedstorage-apiserver and initial implementation (in parallel)
2. Complete and reviewed implementation of sharedstorage-apiserver, in parallel address review comments in the proposal
3. Integration with kube-apiserver
4. Fixing issues needed to ensure higher level integration, so aggregated API servers can be installed with tools such as “helm”.

In the Community Bonding period, I’m planning to create a etcd-gRPC Proxy prototype and to create initial set of issues needed to begin project discussion.

Once we agree how API should look, I’m planning to start working on its implementation. The initial implementation is planned to be done for time of the first evaluation.

The complete API is planned to be done and reviewed by the community before second evaluation. In meanwhile, I’ll be working on first steps needed to integrate it into the kube-apiserver.

For third evaluation, the goal is to have the sharedstorage-apiserver integrated into kube-apiserver, along with issues fixed need to ensure users can install aggregated API server with tools such as helm.

Dates	Tasks
April 23	Community Bonding period begins
April 23 - May 14	etcd-gRPC prototype
April 23 - April 25	Proposal draft. Issues needed to start discussion created.
April 23 - May 14	Start work on sharedstorage-apiserver prototype.
May 14	Community Bonding period ends
May 14 - June 11	Creating initial API implementation and sharing it with the community.
June 11 - June 15	First Evaluation
June 11 - July 9	API completed and reviewed by the community. Initial work on implementing it in the kube-apiserver.

July 9 - July 13	Second Evaluation
July 9 - August 6	API completely integrated into kube-apiserver.
	API integration reviewed by community.
	Working on Certificate generation and related issues to ensure aggregated API servers can be installed via helm.
August 14 - August 21	Final Evaluation

Obligations

During May I'll be able to work full-time (about 40 hours a week). In June, I've exams but I'm targeting to work for about 25 hours a week. In July and August, I don't have any obligations, so I'll be able to work for 40+ hours a week.

General Notes

The key to the successful project is to communicate and coordinate with the community, so the project resolves most of the present problems. Therefore, I'll be taking the following steps:

- Communicate with the community via GitHub and official Slack channel.
- Attend biweekly SIG-API-Machinery meetings, and any other relevant SIG meetings.
- Attend KubeCon EU and if possible other relevant conferences and meetups.
- Keeping mentor informed about my progress on daily basis.
- Create and publish a GitHub repository, containing notes and information about the progress.
- Beside the GitHub repository, I'll create public board for tracking the progress, probably by using GitHub Projects or Trello.
- Maintain a Google Document with daily updates, so it can be easily accessible and shareable by the community.

About me

Who am I

Name: Marko Mudrinić
E-mail: mudrinic.mare@gmail.com
Website and Blog: https://xmudrii.com
GitHub: xmudrii
Slack (Kubernetes): xmudrii
Twitter: xmudrii
University: Faculty of Mathematics, University of Belgrade
Time zone: UTC+01:00 (Central European Time)

Why me

I'm working with Kubernetes for past 8 months. I have experience with deploying applications to Kubernetes, as well as with creating and deploying Kubernetes clusters to cloud providers, including Amazon, Google Cloud Platform and DigitalOcean.

I'm maintainer of Kubicorn project, where I've been focusing on automating the cluster creation process for big variety of cloud providers. I was especially working on getting DigitalOcean support, and you can see some of my work in the following pull requests:

- do: OpenVPN implementation (#217)
- digitalocean: rework digitalocean bootstrap scripts (#527)

I've started contributing to the Kubernetes project itself. I've implemented an integration test for testing the specific case when we're deleting CustomResources with finalizers and then CustomResourceDefinition. For more details, see Pull Request #60592.
Beside this one, I was debugging the issue #57042, and you can the detailed explanation of what I've found in the following comment.
I'll be working on fix for this issue in the upcoming weeks, as well as on fixing another relevant issues.

I have 2 years of experience with the Go programming language, mainly while working on Kubicorn and doctl—command line tool for DigitalOcean services.

Beside coding, I'm an contributor on DigitalOcean Community, where I've published several tutorials, covering topics such as Prometheus and Go. In March 2018, I've been featured as Community DO-er of the month.