SteffenSeckler Typo fixes in device-plugin.md 8d1b2a8 · 8 years ago

504 lines (398 loc) · 19.1 KB

Device Manager Proposal

- References

Authors:

- @RenaudWasTaken - Renaud Gaubert <rgaubert@NVIDIA.com>
- @jiayingz - Jiaying Zhang <jiayingz@google.com>

Motivation

Kubernetes currently supports discovery of CPU and Memory primarily to a minimal extent. Very few devices are handled natively by Kubelet.

It is not a sustainable solution to expect every hardware vendor to add their vendor specific code inside Kubernetes to make their devices usable.

Instead, we want a solution for vendors to be able to advertise their resources to Kubelet and monitor them without writing custom Kubernetes code. We also want to provide a consistent and portable solution for users to consume hardware devices across k8s clusters.

This document describes a vendor independent solution to:

- Discovering and representing external devices
- Making these devices available to the containers, using these devices, scrubbing and securely sharing these devices.
- Health Check of these devices

Because devices are vendor dependent and have their own sets of problems and mechanisms, the solution we describe is a plugin mechanism that may run in a container deployed through the DaemonSets mechanism or in bare metal mode.

The targeted devices include GPUs, High-performance NICs, FPGAs, InfiniBand, Storage devices, and other similar computing resources that require vendor specific initialization and setup.

The goal is for a user to be able to enable vendor devices (e.g: GPUs) through the following simple steps:

- `kubectl create -f http://vendor.com/device-plugin-daemonset.yaml`
- When launching `kubectl describe nodes`, the devices appear in the node status as `vendor-domain/vendor-device`. Note: naming convention is discussed in PR #844

Use Cases

- I want to use a particular device type (GPU, InfiniBand, FPGA, etc.) in my pod.
- I should be able to use that device without writing custom Kubernetes code.
- I want a consistent and portable solution to consume hardware devices across k8s clusters.

Objectives

1. Add support for vendor specific Devices in kubelet:
    - Through an extension mechanism.
    - Which allows discovery and health check of devices.
    - Which allows hooking the runtime to make devices available in containers and cleaning them up.
2. Define a deployment mechanism for this new API.
3. Define a versioning mechanism for this new API.

Non Objectives

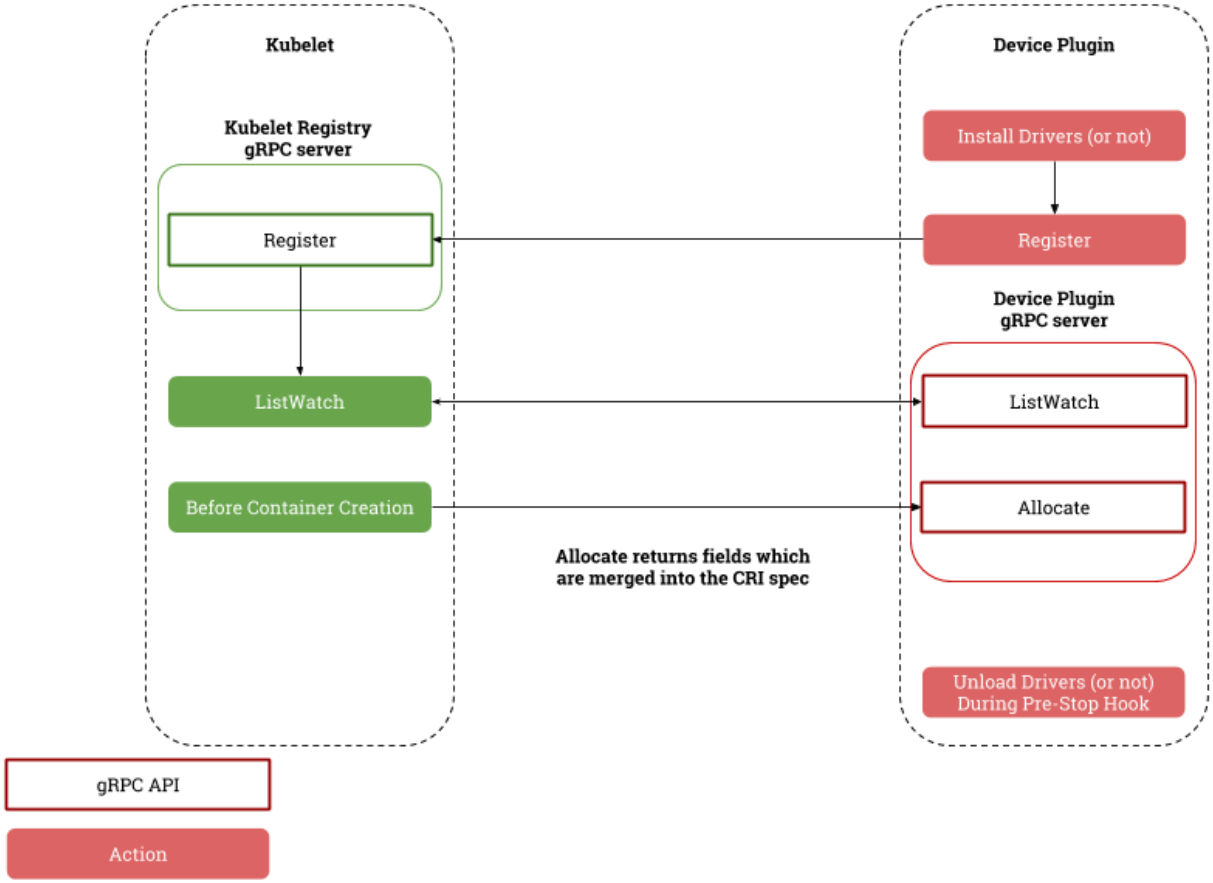
1. Handling heterogeneous nodes and topology related problems
2. Collecting metrics is not part of this proposal. We will only solve Health Check.

TLDR

At their core, device plugins are simple gRPC servers that may run in a container deployed through the pod mechanism or in bare metal mode.

These servers implement the gRPC interface defined later in this design document and once the device plugin makes itself known to kubelet, kubelet will interact with the device through two simple functions:

1. A `ListAndWatch` function for the kubelet to Discover the devices and their properties as well as notify of any status change (device became unhealthy).
2. An `Allocate` function which is called before creating a user container consuming any exported devices



Vendor story

Kubernetes provides to vendors a mechanism called device plugins to:

- advertise devices.
- monitor devices (currently perform health checks).
- hook into the runtime to execute device specific instructions (e.g: Clean GPU memory) and to take in order to make the device available in the container.

```
service DevicePlugin {
    // returns a stream of []Device
    rpc ListAndWatch(Empty) returns (stream ListAndWatchResponse) {}
    rpc Allocate(AllocateRequest) returns (AllocateResponse) {}
}
```

The gRPC server that the device plugin must implement is expected to be advertised on a unix socket in a mounted hostPath (e.g: `/var/lib/kubelet/device-plugins/nvidiaGPU.sock`).

Finally, to notify Kubelet of the existence of the device plugin, the vendor's device plugin will have to make a request to Kubelet's own gRPC server. Only then will kubelet start interacting with the vendor's device plugin through the gRPC apis.

End User story

When setting up the cluster the admin knows what kind of devices are present on the different machines and therefore can select what devices to enable.

The cluster admin knows his cluster has NVIDIA GPUs therefore he deploys the NVIDIA device plugin through: `kubectl create -f nvidia.io/device-plugin.yml`

The device plugin lands on all the nodes of the cluster and if it detects that there are no GPUs it terminates (assuming `restart: OnFailure`). However, when there are GPUs it reports them to Kubelet and starts its gRPC server to monitor devices and hook into the container creation process.

Devices reported by Device Plugins are advertised as Extended resources of the shape `vendor-domain/vendor-device`. E.g., Nvidia GPUs are advertised as `nvidia.com/gpu`

Devices can be selected using the same process as for OIRs in the pod spec. Devices have no impact on QOS. However, for the alpha, we expect the request to have limits == requests.

1. A user submits a pod spec requesting X GPUs (or devices) through `vendor-domain/vendor-device`
2. The scheduler filters the nodes which do not match the resource requests
3. The pod lands on the node and Kubelet decides which device should be assigned to the pod
4. Kubelet calls `Allocate` on the matching Device Plugins
5. The user deletes the pod or the pod terminates

When receiving a pod which requests Devices kubelet is in charge of:

- deciding which device to assign to the pod's containers
- Calling the `Allocate` function with the list of devices

The scheduler is still in charge of filtering the nodes which cannot satisfy the resource requests.

Device Plugin

Introduction

The device plugin is structured in 3 parts:

1. Registration: The device plugin advertises its presence to Kubelet
2. ListAndWatch: The device plugin advertises a list of Devices to Kubelet and sends it again if the state of a Device changes
3. Allocate: When creating containers, Kubelet calls the device plugin's `Allocate` function so that it can run device specific instructions (gpu cleanup, QRNG initialization, ...) and instruct Kubelet how to make the device available in the container.

Registration

When starting the device plugin is expected to make a (client) gRPC call to the `Register` function that Kubelet exposes.

The communication between Kubelet is expected to happen only through Unix sockets and follow this simple pattern:

1. The device plugins sends a `RegisterRequest` to Kubelet (through a gRPC request)
2. Kubelet answers to the `RegisterRequest` with a `RegisterResponse` containing any error Kubelet might have encountered
3. The device plugin start its gRPC server if it did not receive an error

Unix Socket

Device Plugins are expected to communicate with Kubelet through gRPC on an Unix socket. When starting the gRPC server, they are expected to create a unix socket at the following host path: `/var/lib/kubelet/device-plugins/`.

For non bare metal device plugin this means they will have to mount the folder as a volume in their pod spec (see Installation).

Device plugins can expect to find the socket to register themselves on the host at the following path: `/var/lib/kubelet/device-plugins/kubelet.sock`.

Protocol Overview

When first registering themselves against Kubelet, the device plugin will send:

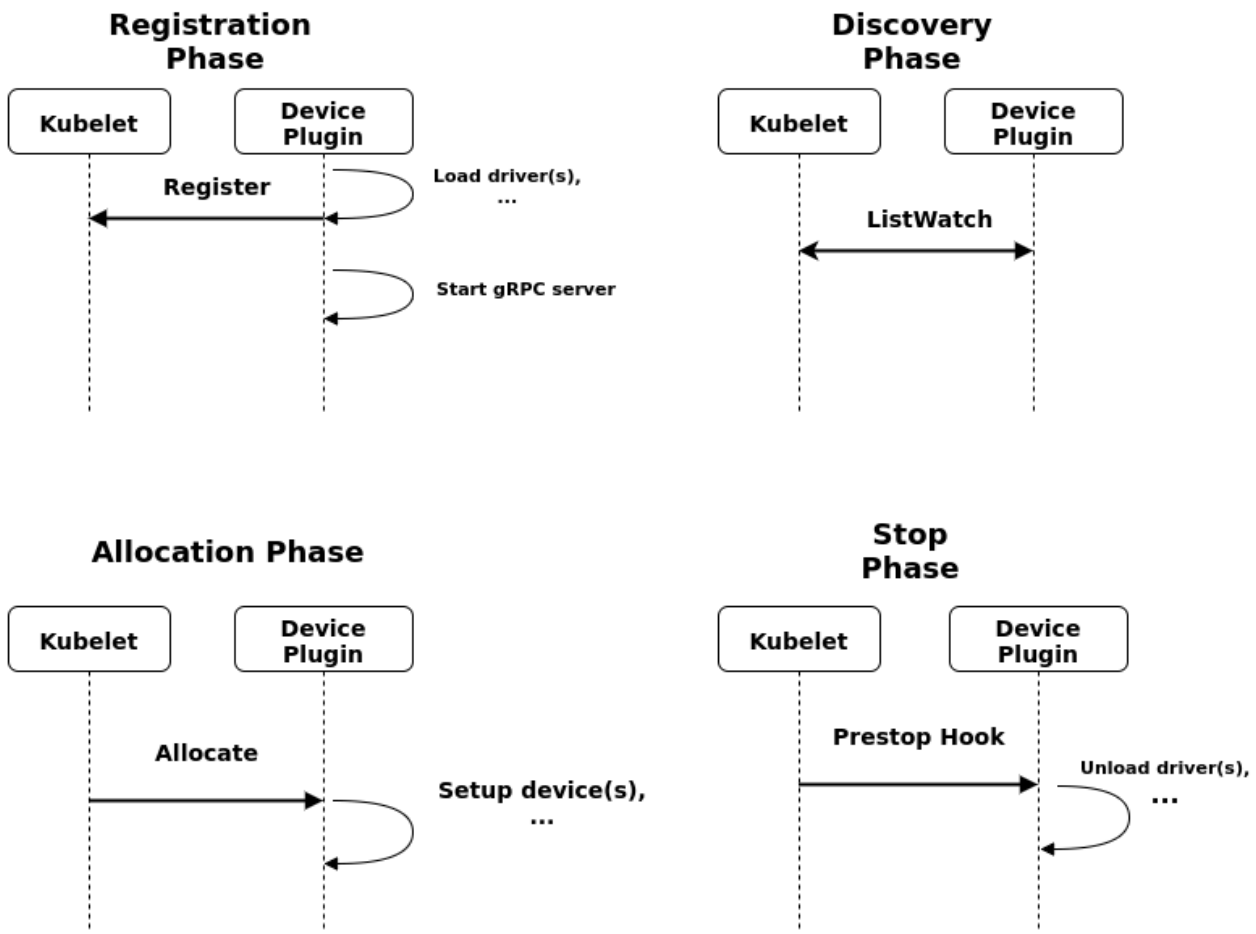
- The name of their unix socket
- The API version against which they were built.
- Their `ResourceName` they want to advertise

Kubelet answers with whether or not there was an error. The errors may include (but not limited to):

- API version not supported
- A device plugin already registered this `ResourceName`

After successful registration, Kubelet will interact with the plugin through the following functions:

- ListAndWatch: The device plugin advertises a list of Devices to Kubelet and sends it again if the state of a Device changes
- `Allocate`: Called when creating a container with a list of devices



API Specification

```
// Registration is the service advertised by the Kubelet
// Only when Kubelet answers with a success code to a Register Request
// may Device Plugins start their service
// Registration may fail when device plugin version is not supported by
// Kubelet or the registered resourceName is already taken by another
// active device plugin. Device plugin is expected to terminate upon registr
service Registration {
    rpc Register(RegisterRequest) returns (Empty) {}
}

// DevicePlugin is the service advertised by Device Plugins
service DevicePlugin {
    // ListAndWatch returns a stream of List of Devices
    // Whenever a Device state change or a Device disappears, ListAndWatch
    // returns the new list
    rpc ListAndWatch(Empty) returns (stream ListAndWatchResponse) {}
```

```
    // Allocate is called during container creation so that the Device
    // Plugin can run device specific operations and instruct Kubelet
    // of the steps to make the Device available in the container
    rpc Allocate(AllocateRequest) returns (AllocateResponse) {}
}

message RegisterRequest {
    // Version of the API the Device Plugin was built against
    string version = 1;
    // Name of the unix socket the device plugin is listening on
    // PATH = path.Join(DevicePluginPath, endpoint)
    string endpoint = 2;
    // Schedulable resource name
    string resource_name = 3;
}

// - Allocate is expected to be called during pod creation since allocation
//   failures for any container would result in pod startup failure.
// - Allocate allows kubelet to exposes additional artifacts in a pod's
//   environment as directed by the plugin.
// - Allocate allows Device Plugin to run device specific operations on
//   the Devices requested
message AllocateRequest {
    repeated string devicesIDs = 1;
}

// Failure Handling:
// if Kubelet sends an allocation request for dev1 and dev2.
// Allocation on dev1 succeeds but allocation on dev2 fails.
// The Device plugin should send a ListAndWatch update and fail the
// Allocation request
message AllocateResponse {
    repeated DeviceRuntimeSpec spec = 1;
}

// ListAndWatch returns a stream of List of Devices
// Whenever a Device state change or a Device disappears, ListAndWatch
// returns the new list
message ListAndWatchResponse {
    repeated Device devices = 1;
}

// The list to be added to the CRI spec
message DeviceRuntimeSpec {
    string ID = 1;

    // List of environment variable to set in the container.
    map<string, string> envs = 2;
    // Mounts for the container.
```

```
    repeated Mount mounts = 3;
    // Devices for the container
    repeated DeviceSpec devices = 4;
}

// DeviceSpec specifies a host device to mount into a container.
message DeviceSpec {
    // Path of the device within the container.
    string container_path = 1;
    // Path of the device on the host.
    string host_path = 2;
    // Cgroups permissions of the device, candidates are one or more of
    // * r - allows container to read from the specified device.
    // * w - allows container to write to the specified device.
    // * m - allows container to create device files that do not yet exist.
    string permissions = 3;
}

// Mount specifies a host volume to mount into a container.
// where device library or tools are installed on host and container
message Mount {
    // Path of the mount on the host.
    string host_path = 1;
    // Path of the mount within the container.
    string mount_path = 2;
    // If set, the mount is read-only.
    bool read_only = 3;
}

// E.g:
// struct Device {
//    ID: "GPU-fef8089b-4820-abfc-e83e-94318197576e",
//    State: "Healthy",
//}
message Device {
    string ID = 2;
    string health = 3;
}
```

HealthCheck and Failure Recovery

We want Kubelet as well as the Device Plugins to recover from failures that may happen on any side of this protocol.

At the communication level, gRPC is a very strong piece of software and is able to ensure that if failure happens it will try its best to recover through exponential backoff reconnection and Keep Alive checks.

The proposed mechanism intends to replace any device specific handling in Kubelet. Therefore in general, device plugin failure or upgrade means that Kubelet is not able to accept any pod requesting a Device until the upgrade or failure finishes.

If a device fails, the Device Plugin should signal that through the `ListAndWatch` gRPC stream. We then expect Kubelet to fail the Pod.

If any Device Plugin fails the behavior we expect depends on the task Kubelet is performing:

- In general we expect Kubelet to remove any devices that are owned by the failed device plugin from the node capacity. We also expect node allocatable to be equal to node capacity.
- We however do not expect Kubelet to fail or restart any pods or containers running that are using these devices.
- If Kubelet is in the process of allocating a device, then it should fail the container process.

If the Kubelet fails or restarts, we expect the Device Plugins to know about it through gRPC's Keep alive feature and try to reconnect to Kubelet.

When Kubelet fails or restarts it should know what are the devices that are owned by the different containers and be able to rebuild a list of available devices. We are expecting to implement this through a checkpointing mechanism that Kubelet would write and read from.

API Changes

When discovering the devices, Kubelet will be in charge of advertising those resources to the API server as part of the kubelet node update current protocol.

We will be using extended resources to schedule, trigger and advertise these Devices. When a Device plugin registers two `foo-device` the node status will be updated to advertise 2 `vendor-domain/foo-device`.

If a user wants to trigger the device plugin he only needs to request this through the same mechanism as OIRs in his Pod Spec.

Upgrading your cluster

TLDR: Given that we cannot guarantee that the Device Plugins are not running a daemon providing a critical service to Devices and when stopped will crash the running containers, it is up to the vendor to specify the upgrading scheme of their device plugin.

However, If you are upgrading either Kubelet or any device plugin the safest way is to drain the node of all pods and upgrade.

Depending on what you are upgrading and what changes happened then it is completely possible to only restart just Kubelet or just the device plugin.

Upgrading Kubelet

This assumes that the Device Plugins running on the nodes fully implement the protocol and are able to recover from a Kubelet crash.

Then, as long as the Device Plugin API does not change upgrading Kubelet can be done seamlessly through a Kubelet restart.

Currently: As mentioned in the Versioning section, we currently expect the Device Plugin's API version to match exactly the Kubelet's Device Plugin API version. Therefore if the Device Plugin API version change then you will have to change the Device Plugin too.

Future: When the Device Plugin API becomes a stable feature, versioning should be backward compatible and even if Kubelet has a different Device Plugin API,

it should not require a Device Plugin upgrade.

Refer to the versioning section for versioning scheme compatibility.

Upgrading Device Plugins

Because we cannot enforce what the different Device Plugins will do, we cannot say for certain that upgrading a device plugin will not crash any containers on the node.

It is therefore up to the Device Plugin vendors to specify if the Device Plugins can be upgraded without impacting any running containers.

As mentioned earlier, the safest way is to drain the node before upgrading the Device Plugins.

Installation

The installation process should be straightforward to the user, transparent and similar to other regular Kubernetes actions. The device plugin should also run in containers so that Kubernetes can deploy them and restart the plugins when they fail. However, we should not prevent the user from deploying a bare metal device plugin.

Deploying the device plugins through DemonSets makes sense as the cluster admin would be able to specify which machines it wants the device plugins to run on, the process is similar to any Kubernetes action and does not require to change any parts of Kubernetes.

Additionally, for integrated solutions such as `kubeadm` we can add support to auto-deploy community vetted Device Plugins. Thus not fragmenting once more the Kubernetes ecosystem.

For users installing Kubernetes without using an integrated solution such as `kubeadm` they would use the examples that we would provide at:
`https://github.com/vendor/device-plugin/tree/master/device-plugin.yaml`

YAML example:

```yaml
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
spec:
    template:
        metadata:
            labels:
                - name: device-plugin
        spec:
            containers:
                name: device-plugin-ctr
                image: NVIDIA/device-plugin:1.0
                volumeMounts:
                  - mountPath: /device-plugin
                  - name: device-plugin
           volumes:
             - name: device-plugin
               hostPath:
                   path: /var/lib/kubelet/device-plugins
```

Versioning

Currently we require exact version match between Kubelet and Device Plugin. API version is expected to be increased only upon incompatible API changes.

Follow protobuf guidelines on versioning:

- Do not change ordering
- Do not remove fields or change types
- Add optional fields
- Introducing new fields with proper default values
- Freeze the package name to `apis/device-plugin/v1alpha1`
- Have kubelet and the Device Plugin negotiate versions if we do break the API

References

- Adding a proposal for hardware accelerators
- Enable "kick the tires" support for NVIDIA GPUs in COS
- Extend experimental support to multiple NVIDIA GPUs
- Kubernetes Meeting notes
- Better Abstraction for Compute Resources in Kubernetes
- Extensible support for hardware devices in Kubernetes (join Kubernetes-dev@googlegroups.com for access)