[YARN-8851] YARN's New Device Plugin Framework Design Proposal

Zhankun Tang, Wangda Tan

Revisions **1**

Motivation **2**

Objectives **3**

Phase 1 3

Phase 2 3

Phase 3 3

User Story **3**

For administrator 3

For vendor developers 4

For end user 4

Design and Implementation **4**

Plugin Interfaces 5

Plugin package and deploy 5

Adapter Layer 5

Interaction between plugin and NM 6

NM API to query resource allocation 6

Future Work **6**

References: **7**

Revisions

Version	Changes	Date
0.2	The new plugin framework design	2018-10-14
0.1	Initial design and issues discussed	2018-09-14

Motivation

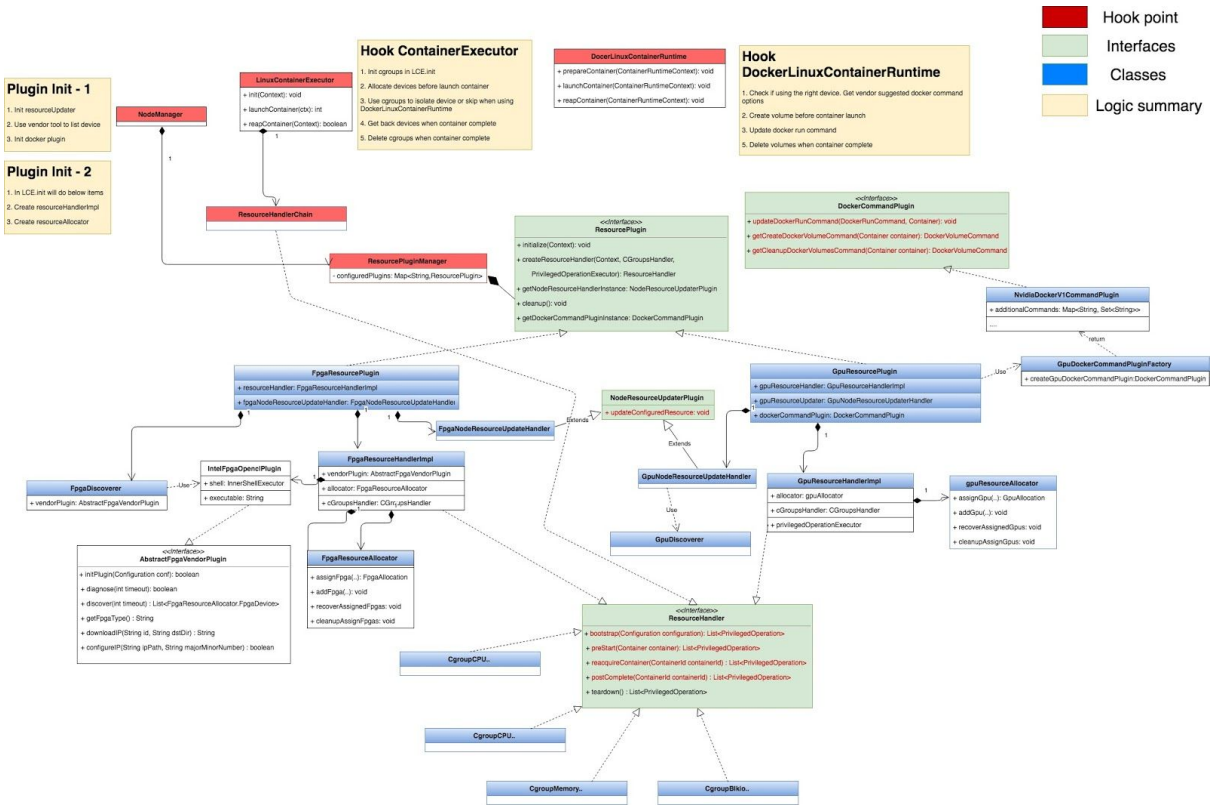

Figure 1. Current device plugin framework UML

When someone wants to add a new resource type backed by vendor plugin, he/she should follow the above existing YARN resource plugin framework. Which means, get the plugin code done and put it into YARN NM code base. But that is not an easy task and adds maintainability burdens for both YARN community and the vendor:

1. It requires deep knowledge of YARN internals for vendor developer
   Check above figure. The blue color is what a developer need to implement. At least 6 classes to be implemented (If you wanna support Docker, you'll implement one more "DockerCommandPlugin"). More painful thing is that when implementing the "ResourceHandler" interface, the developer must know the YARN NM internal concepts like container launch mechanism, cgroups operations, docker runtime operations. And if one wants isolation, the native container-executor also need a new module written in C language
2. Any changes to the core YARN components could potentially affect all plugins
3. Duplicate resource allocator logics may be implemented again and again. We could make a common one

Based on above reasons and in order for YARN and vendor specific plugin to evolve independently, we propose a way to extend existing resource plugin framework into a new one.

Objectives

Phase 1

- Simplified scalable Java APIs for vendor developer to use and an example device plugin in YARN
- Real Vendor plugin code should not be put into YARN code base
- Vendor plugin class loaded into NM process when NM starts
- Support new extended resource discovery, allocation, allocation query NM REST API in new device plugin framework
- No affect on existing GPU/FPGA resource

Phase 2

- Support resource isolation in both cgroups and Docker way
- Support aggregated resource allocation query RM APIs
- Support topology scheduling
- Move existing GPU/FPGA plugin into new device plugin framework

Phase 3

- Support long-running gRPC server of device plugin

User Story

For administrator

1. We add two new configurations here. Enable extended vendor device plugin in yarn-site.xml:
   <!-- Enable new device plugin framework -->
   <property>
   <name>yarn.nodemanager.resource-plugins.enable-extended-device</name>
   <value>true</value>
   </property>

   <!-- Here the "com.cmp1.hardware1.plugin" will be used to create plugin instance-->
   <property>
   <name>yarn.nodemanager.resource-plugins.extended</name>
   <value>com.company1.hardware1.plugin</value>
   </property>

2. Define the new resource in resource-type.xml with the resource name plugin wants to register. For instance:
   <configuration>
   <property>
   <name>yarn.resource-types</name>
   <value>cmp1.com/hardware1</value>
   </property>
   </configuration>
3. Get the device plugin release jar file and put it into CLASSPATH or under directory ${HADOOP_YARN_HOME}/share/hadoop/yarn in order for the jar to be loaded
4. Start YARN

For vendor developers

Maintains an official YARN device plugin Maven project and implement above 4 interfaces based on YARN device plugin library.

For end user

Request new resource as usual. In above example, request “cmp1.com/hardware1” in container request

Design and Implementation

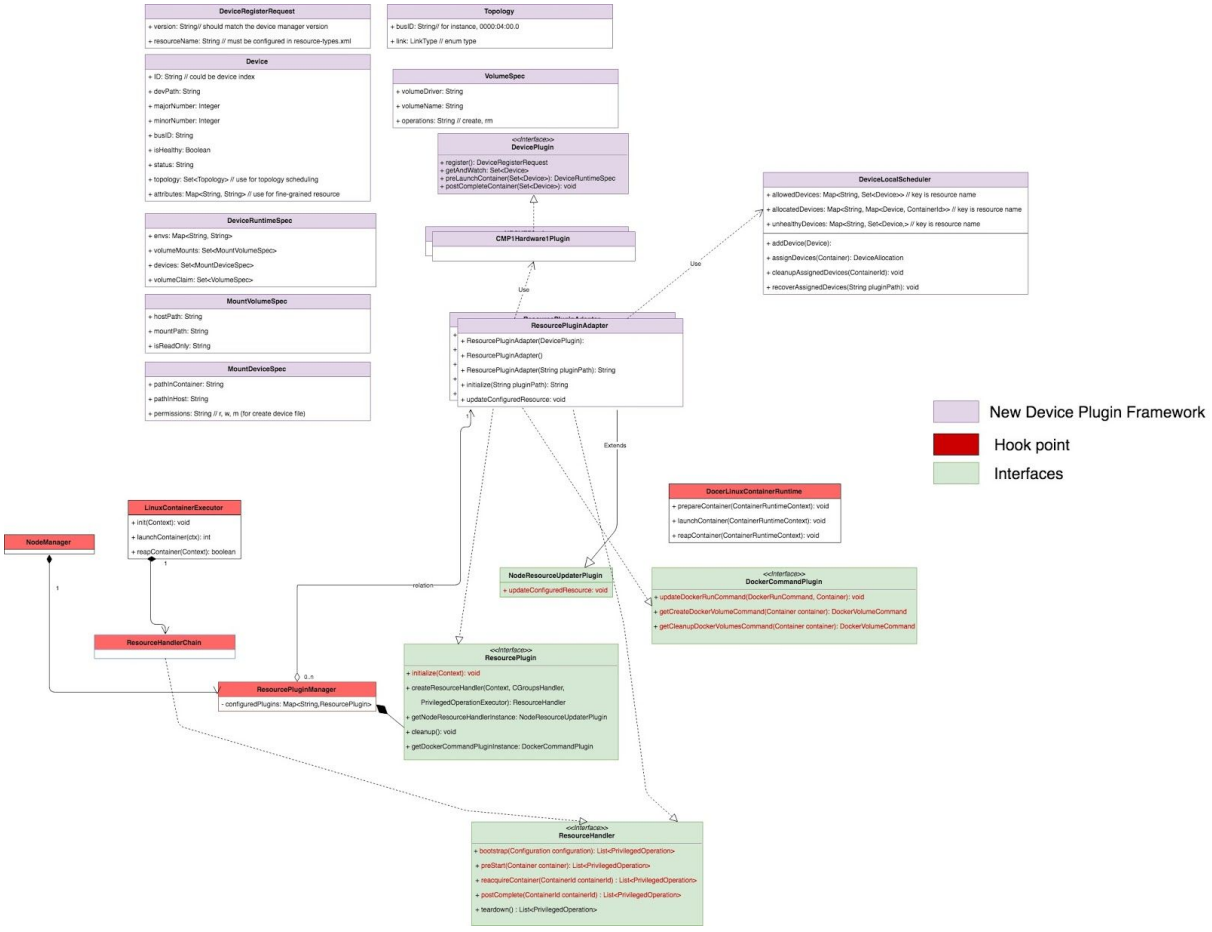

Figure 2. New Device Plugin Framework

The purple color in above figure illustrates the new device plugin framework we want to add which has little changes to existing framework. It mainly consists two parts. The interfaces for the vendor to implement and the adapter to adapt the new plugin interfaces to existing device plugin framework.
With this adapter design pattern involved, the vendor developer wouldn't manipulate YARN internal components but just the plugin library we provided.

Plugin Interfaces

Let's go through the new interfaces that a vendor plugin should implement.

1. *register(): DeviceRegisterRequest*
2. *getAndWatch: Set<Device>*
3. *preLaunchContainer(Set<Device>): DeviceRuntimeSpec*
4. *postCompleteContainer(Set<Device>): void*

The ***register*** interface is used for the vendor plugin to advertise a new resource type to "ResourcePluginManager" and then the ResourceManager. The "DeviceRegisterRequest"

returned by the method consists a version for “ResourcePluginManager” to do the compatibility check and a resource type name like “cmp1.com/hardware1”. Note that the resource type name must be pre-defined in the “resource-type.xml”.
The ***getAndWatch*** interface is used to get latest vendor device list. The resource count pre-defined in node-resources.xml will be override. So it’s recommended that the vendor plugin manages available devices by itself.
The ***preLaunchContainer*** interface is used before a container launch. The NM invoke this interface to let the plugin do some preparation work like create volume before container launch and give hints on how to expose the devices to container when launch it.
The ***postCompleteContainer*** interface is used for the plugin to do some cleanup work after container finish.

The source code of the above API library could be put under “org.apache.hadoop.yarn.server.nodemanager.api” as library to be used as Maven dependency for vendor plugin project.

Plugin package and deploy

The vendor plugin project is a seperate project and will expose a jar file for YARN NM to load needed plugin classes when bootstrap. The admin only need to put the jar file into CLASSPATH or YARN’s jar folder.

Adapter Layer

The adapter class will implement multiple interfaces which are “NodeResourceUpdaterPlugin”, “ResouceHandler”, “ResourcePlugin” and “DockerCommandPlugin”. Each adapter class will be initialized with a plugin instance passed in and adapt calls to the plugin interfaces to discover resource and prepare launch container. It will translate the message got from plugin to the YARN internal components operations.

Interaction between plugin and NM

We’ll reuse the existing hook points for the adapter to intercept NM calls and co-operate plugin APIs. A basic procedure in NM is as below:

1. When NM starts, “ResourcePluginManager” will load the plugin class from configuration.
   The instance will be used to new a adapter instance which will be stored in “configuredPlugins”. And adapter’s initialization will invoke plugin registration.
   Besides, in “LinuxContainerExecutor” initialization, the adapter’s bootstrap method will be invoked which populate device resource and stored in the “DeviceLocalScheduler”.
   Then when “NodeStateUpdater” service start, it will call the adapter to update the “Resource” object which will be sent to “ResourceManager”.
2. When a container comes before launch, the “LinuxContainerExecutor” will invoke adapter’s preStart method which will in turn ask “DeviceLocalScheduler” to allocate

device for this container. After device allocation, the collection of devices will be passed to the plugin's preLaunchContainer to get back the DeviceRuntimeSpec. Then it will process based on the container runtime.

3. When a container finish, the adapter's postComplete will be invoked, and handling will be passed to "DeviceLocalScheduler" and the plugin to do cleanup.

NM API to query resource allocation

To query the status/allocation of device resource, the URI of the API:

nodemanager_address:port/ws/v1/node/resources/{resource_name}

Future Work

Topology Scheduling
For device topology scheduling, we'll use efficient data structure in "DeviceLocalScheduler" to store the devices for scheduling. In the plugin library, we'll define known common link type for plugin to specify. For instance, we have below link types based on Nvidia GPU's topology definition. Some are hard to understand, so we change to simpler version. Below are our topology definition. From high speed IO to low:

- P2PLinkNVLink
- P2PLinkSameCPU
  Nvidia originally means same host bridge. But host bridge is hard to understand because it has been packaged to CPU (Same NUMA node)
- P2PLinkCrossCPU
  Cross CPU through socket-level link (e.g. QPI). Usually cross NUMA node
- P2PLinkSingleSwitch
  Just need to traverse one PCIe switch
- P2PLinkMultiSwitch
  Need to traverse multiple PCIe switch

Independent Long-Running Plugin
Instead of loading plugin code into NM's process, to be more secure, we may would like to run the plugin as a long running RPC server, we could easily extend our adapter to be a RPC client to communicate with it.

Refactor Existing GPU/FPGA Plugin
Once the new device plugin become mature, we could consider refactor current GPU/FPGA device plugin into this framework.

References:

K8s device manager
https://github.com/kubernetes/community/blob/master/contributors/design-proposals/resource-management/device-plugin.md

K8s device plugins
https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/

K8s CRI
https://kubernetes.io/blog/2018/05/24/kubernetes-containerd-integration-goes-ga/
https://github.com/kubernetes/kubernetes/blob/master/pkg/kubelet/apis/cri/runtime/v1alpha2/api.proto

K8s GPU device plugin
https://github.com/nvidia/nvidia-container-runtime#environment-variables-oci-spec

Nvdia GPU topology
https://github.com/NVIDIA/nvidia-docker/blob/1.0/src/nvml/nvml.go#L80
https://raw.githubusercontent.com/NVIDIA/nvidia-settings/168e17f53098254b4a5ab93eeb2f23c80ca1d97f/src/nvml.h (search “nvmlGpuLevel_enum”)

Microsoft KubeGPU
https://github.com/Microsoft/KubeGPU/blob/master/plugins/nvidiagpuplugin/gpu/nvidia/nvidia_gpu_manager.go#L170

YARN-313 Add Admin API for supporting node resource configuration in command line
YARN-5592 Dynamically adding resource types
YARN-7119 Support multiple resource types in rmadmin updateNodeResource command
YARN-5983 FPGA as a first-class citizen
YARN-6223 GPU as a first-class citizen