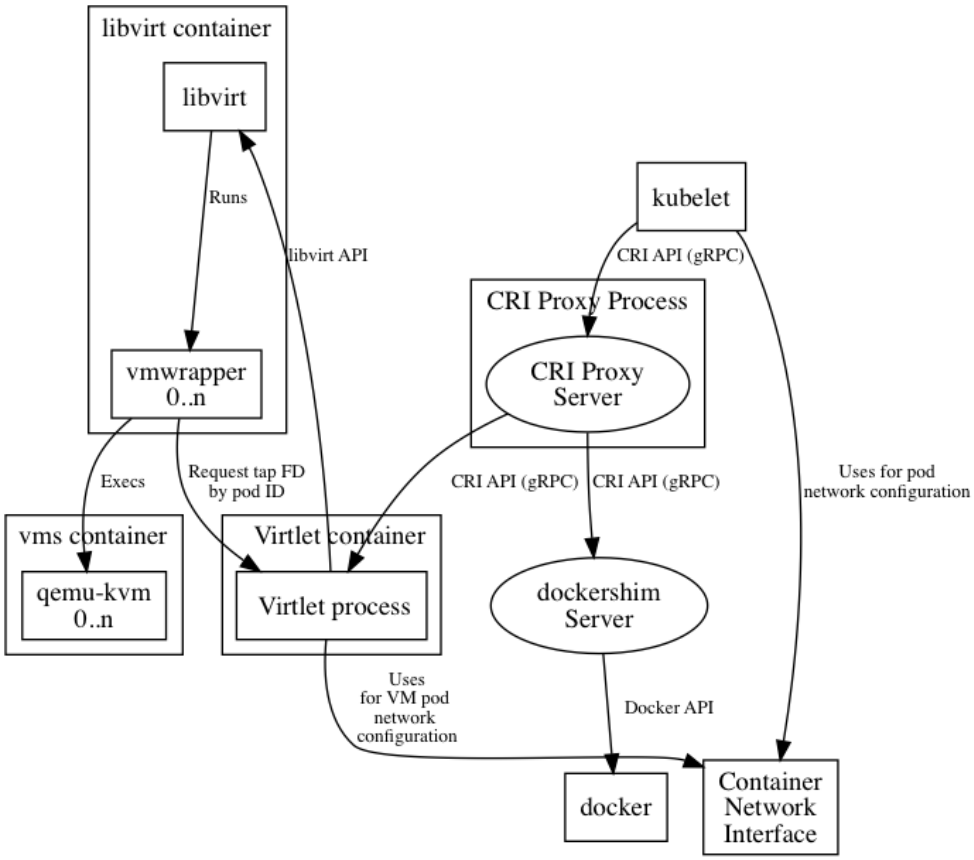
QAT enabling in Virtlet

Virtlet

what is Virtlet

Virtlet is a Kubernetes CRI (Container Runtime Interface) implementation for running VM-based pods on Kubernetes clusters. (CRI is what enables Kubernetes to run non-Docker flavors of containers, such as Rkt.) For the sake of simplicity of deployment, Virtlet itself runs as a DaemonSet, essentially acting as a hypervisor and making the **CRI proxy** (Provides the possibility of mixing docker-shim and VM based workloads on the same k8s node) available to run the actual VMs This way, it's possible to have both Docker and non-Docker pods run on the same node.



Components

- Virtlet manager: Implements the CRI interface for virtualization and image handling
- Libvirt: The standard instance of libvirt for KVM.
- vmwrapper: Responsible for preparing the environment for the emulator

- Emulator: Currently qemu with KVM support (with possibility of disabling KVM for nested virtualization tests)
- ...

VM Pod Lifecycle

Startup

- A VM pod is created in Kubernetes cluster.
- Scheduler places the pod on a node based on the requested resources (CPU, memory, etc.) as well as pod's nodeSelector and pod/node affinity constraints, taints/tolerations and so on.
- Kubelet running on the target node accepts the pod.
- Kubelet invokes a CRI call RunPodSandbox to create the pod sandbox which will enclose all the containers in the pod definition.
- If there's a Virtlet-specific annotation Kubernetes.io/target-runtime: virtlet.cloud, CRI proxy passes the call to Virtlet.
- Virtlet saves sandbox metadata in its internal database, sets up the network namespace and then uses internal tapmanager mechanism to invoke ADD operation via the CNI plugin as specified by the CNI configuration on the node.
- The CNI plugin configures the network namespace by setting up network interfaces, IP addresses, routes, iptables rules and so on, and returns the network configuration information to the caller as described in the CNI spec.
- Virtlet's tapmanager mechanism adjusts the configuration of the network namespace to make it work with the VM.
- After creating the sandbox, Kubelet starts the containers defined in the pod sandbox. Currently, Virtlet supports just one container per VM pod. So, the VM pod startup steps after this one describe the startup of this single container.
- Depending on the image pull police of the container, Kubelet checks if the image needs to be pulled by means of ImageStatus call and then uses PullImage CRI call to pull the image if it doesn't exist or if imagePullPolicy: Always is used.
- If PullImage is invoked, Virtlet resolves the image location based on the image name translation configuration, then downloads the file and stores it in the image store.
- After the image is ready (no pull was needed or the PullImage call completed successfully), Kubelet uses CreateContainer CRI call to create the container in the pod sandbox using the specified image.
- Virtlet uses the sandbox and container metadata to generate libvirt domain definition, using vmwrapper binary as the emulator and without specifying any network configuration in the domain.
- After CreateContainer call completes, Kubelet invokes StartContainer call on the newly created container.
- Virtlet starts the libvirt domain. libvirt invokes vmwrapper as the emulator, passing it the necessary command line arguments as well as environment variables set by Virtlet.
- vmwrapper uses the environment variable values passed to Virtlet to communicate with tapmanager over an Unix domain socket, retrieving a file descriptor for a tap device and/or pci address of SR-IOV device set up by tapmanager.
- tapmanager uses its own simple protocol to communicate with vmwrapper because it needs to send file descriptors over the socket.
- vmwrapper then updates the command line arguments to include the network interface information and execs the actual emulator (qemu).

Delete

- Kubelet notices the pod being deleted.
- Kubelet invokes StopContainer CRI calls which is getting forwared to Virtlet based on the containing pod sandbox annotations.
- Virtlet stops the libvirt domain. libvirt sends a signal to qemu, which initiates the shutdown. If it doesn't quit in a reasonable time determined by pod's termination grace period, Virtlet will forcibly terminate the domain, thus killing the qemu process.
- After all the containers in the pod (the single container in case of Virtlet VM pod) are stopped, Kubelet invokes StopPodSandbox CRI call.
- Virtlet asks its tapmanager to remove pod from the network by means of CNI DEL command.
- After StopPodSandbox returns, the pod sandbox will be eventually GC'd by Kubelet by means of RemovePodSandbox CRI call.
- Upon RemovePodSandbox, Virtlet removes the pod metadata from its internal database.

Virtlet is used to create a virtual machine to support some necessary features needed by ICN. In ICN use case we need IpSec to finish some functions. So using QAT devices to speed up the connections is important. But after tests, I found that virtlet doesn't recognize the qat vf device.

Gaps

- Virtlet considers all other devices bound vfio-pci drivers as a volume device and adds them into libvritxml as block disk type with disk driver. This will cause vm startup errors.
- Virtlet binds the network devices after the creation of libvirt domain file, and its default hostdev id number starts from 0, it will make conflict when we add other type device to libvirt domain file by pci-passthrough
- Virtlet can not recognize other sriov device
- ...

To solve these problems, we should first have a clear knowledge of device plugin. A related concept for device plugin is Kubernetes extended-resources. In conclusion, By sending a patch node request to the Kubernetes apiserver, a custom resource type is added to the node, which is used for the quota statistics of the resource and the corresponding QoS configuration.

Example

To send a patch node request conveniently, start a proxy, so that you can easily send requests to the Kubernetes API server, we first execute *kube proxy* c ommand to start it temporarily, then add six *intel.com/devices* resource to a node (**~1** *in the commands will automatically transform into* **/**):

```
curl --header "Content-Type: application/json-
patch+json" \
--request PATCH \
--data '[{"op": "add", "path": "/status/capacity
/intel.com~1devices", "value": "6"}]' \
http://localhost:8001/api/v1/nodes/<your-node-name>
/status
```

Now we extend 6 *intel.com/devices* resources for your node, then we can see

```
kubectl describe node xxx
...
Capacity:
  ephemeral-storage:
3650656984Ki
  cpu:                72
  memory:
263895388Ki
  intel.com/devices:   6
  pods:
                110
...
```

Now we can use these resources in our pod by adding *intel.com/devices: "1"* to *spec.containers.resources.requests/limits* and the pod will be scheduled with statistics.

To clean up the extended resources, execute the following commands:

```
curl --header "Content-Type: application/json-
patch+json" \
--request PATCH \
--data '[{"op": "remove", "path": "/status/capacity
/intel.com~1devices"}]' \
http://localhost:8001/api/v1/nodes/<your-node-name>
/status
```

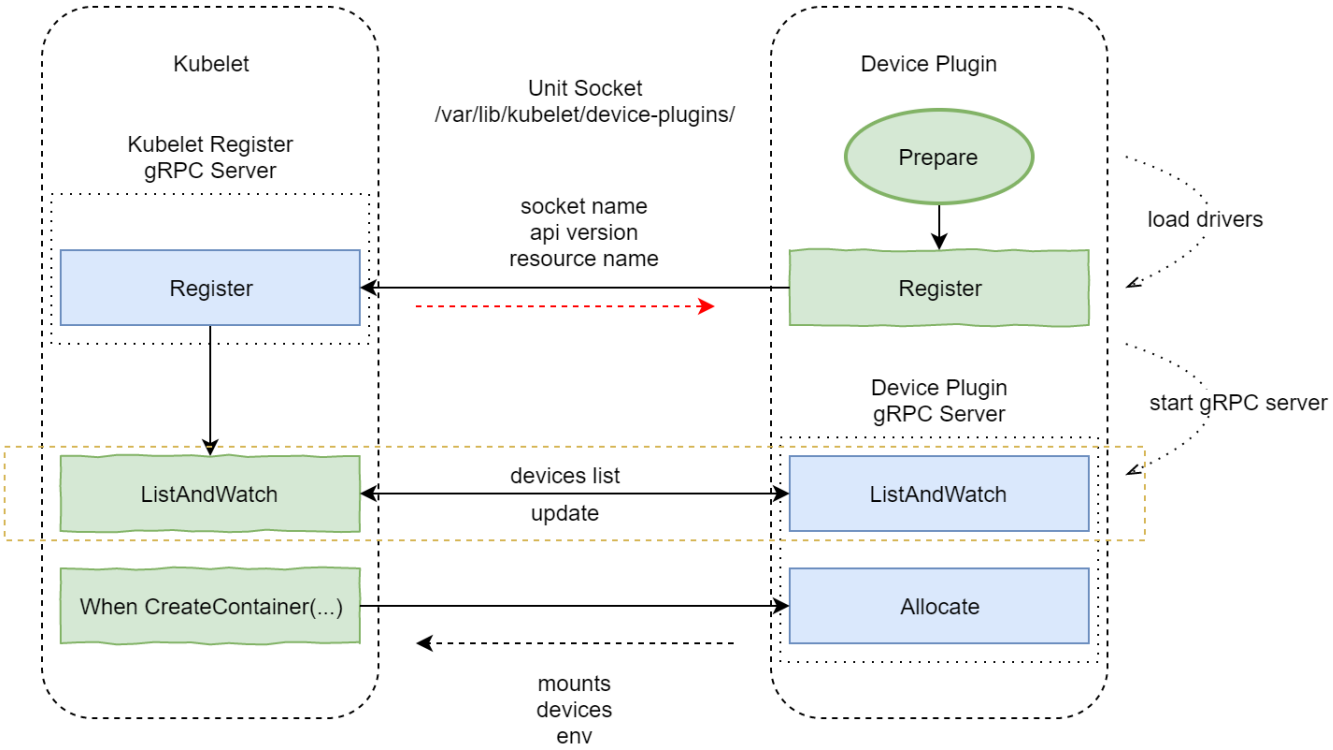
Device plugin

Overview

Kubernetes provides to vendors a mechanism called device plugins to finish the following three tasks, device plugins are simple gRPC servers that may run in a container deployed through the pod mechanism or in bare metal mode.

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

- 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.



Why device plugin

- Very few devices are handled natively by Kubelet (cpu and memory)
- Need a sustainable solution for vendors to be able to advertise their resources to Kubelet and monitor them without writing custom Kubernetes code
- A consistent and portable solution to consume hardware devices across k8s clusters to use a particular device type (GPU, QAT, FPGA, etc.) in pods
- ...

How it works

In Kubernetes, Kubelet will offer a register gRPC server which allows the device plugin to register itself to Kubelet. When starting, the device plugin will make a (client) gRPC call to the Register function that Kubelet exposes. The device plugins sends a RegisterRequest to Kubelet to notify Kubelet of the following informations, and Kubelet answers to the RegisterRequest with a RegisterResponse containing any error Kubelet might have encountered (api version not supported, resource name already register), then the device plugin start its gRPC server if it did not receive an error.

1. Its own unix socket name, which will receive the requests from Kubelet through the gRPC apis.
2. The api version of device plugin itself
3. The resource name they want to advertise. The resource name must follow a specified format (vendor-domain/vendor-device). such as *intel.com /qat*

After successful registration, Kubelet will call the *ListAndWatch* function from the device plugin. A *ListAndWatch* function is for the Kubelet to Discover the devices and their properties as well as notify of any status change (devices become unhealthy). The list of devices is returned as an array of all devices description information (ID, health status) of the resource. Kubelet records this resource and its corresponding number of devices to *node.status.capacity /allocable* and updates it to apiserver. This function will always loop check, once the device is abnormal or unplugged from the machine, it will update and return the latest device list to Kubelet.

In this way, when creating a pod, fields such as *intel.com/qat* can be added to spec.containers.resource.limits/requests: "1" to inform Kubernetes to schedule the pod to nodes with more than one *intel.com/qat* resource allowance. When the pod is to run, Kubelet will call device plugin *allocate* function. Device plugin may do some initialization operations, such as QAT configuration or QRNG initialization. If initialization is successful, this function will return how to config the device assigned to the pod when the container is created, and this configuration will be passed to the container runtime as a parameter used to run the container.

User Flow

To use the extended resource, we add *intel.com/qat* to spec.containers.resource.limits/requests, we expect the request to have limits == requests.

1. A user submits a pod spec requesting 1 QAT through *intel.com/qat*

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

Enable QAT supported by virtlet

Gaps detection in source code

When testing the QAT sr-iov support condition with the officer virtlet image, together with QAT device plugin. We take thie simple straightforward method that adds the resource name *qat.intel.com/generic* advertised by the QAT device plugin to fields spec.containers.resource.limits and spec.containers.resource.requests with value "1". It works correctly in plain kubernetes pods. But in a virtlet vm pod, we encountered the conflict caused by the configuration transformed between virtual machine and pod by virtlet. The issues is that when allocating a QAT vf device to virtlet vm pod, Kubelet will add the extended device to *kubeapi.PodSandboxConfig.Devices (k8s.io/kubernetes/pkg/kubelet/apis/cri/runtime/v1alpha2 - v1.14).* Then virtlet will incorrectly transform all these devices to its volume devices and considers them as block disk with disk drivers bound to them later.

```
for _, dev := range in.Config.Devices {
    r.VolumeDevices = append(r.VolumeDevices, types.VMVolumeDevice{
        DevicePath: dev.ContainerPath,
        HostPath:   dev.HostPath,
    })
}
```

It causes the errors that too many disks, disks' reading issues, denied permission and so on after a vm pod starts. And regardless of this, I want to assign QAT vf to the virtlet pod by pci-passthrough. So I want to add corresponding fields into the libvirt instance domain xml created by virtlet. After code analysis, virtlet is a cri implement and in its createDomain(config *types.VMConfig) *libvirtxml.Domain *(pkg/libvirttools/virtualization.go)* I detect the xml file creation and find it is using the *libvirtxml "github.com/libvirt/libvirt-go-xml"* go module. So the whole workflow is clear now and I can fix it then.

```
domain := &libvirtxml.Domain{
    Devices: &libvirtxml.DomainDeviceList{
        Emulator: "/vmwrapper",
        Inputs: []libvirtxml.DomainInput{
            {Type: "tablet", Bus: "usb"},
        },
        Graphics: []libvirtxml.DomainGraphic{
            {VNC: &libvirtxml.DomainGraphicVNC{Port: -1}},
        },
        Videos: []libvirtxml.DomainVideo{
            {Model: libvirtxml.DomainVideoModel{Type: "cirrus"}},
        },
        Controllers: []libvirtxml.DomainController{
            {Type: "scsi", Index: &scsiControllerIndex, Model: "virtio-scsi"},
        },
    },

...
```

Key Point

Because Virtlet creates a VM by libvirt instance. So we config QAT devices to its domain file to finish the QAT device assignment. Virtlet can get the QAT device id from the environment variables which are advertised by QAT device plugin and passed by Kubelet. Then we can easily assign a QAT vf device into a Virtlet VM by PCI-passthrough supported by libvirt hostdev api.

```
<hostdev mode='subsystem' type='pci' managed='yes'>
    <driver name='vfio'/>
    <source>
      <address domain='0x0000' bus='0x3d' slot='0x02' function='0x2'/>
    </source>
    <alias name='hostdev0'/>
    <address type='pci' domain='0x0000' bus='0x00' slot='0x04' function='0x0'/>
</hostdev>
```

And for further code information, you can get from the my fork version of Virtlet in https://github.com/leyao-daily/virtlet

Example

I have uploaded the QAT enabled image into docker hub and you can download it by '**docker pull integratedcloudnative/virtlet-qat:test**'. After the Virtlet Pod runs, you can set up a VM with QAT vf device. Add the orange line with the number of QAT vf you want to assign into spec.containers.resource.limits /requests of your Virtlet VM yaml file like below.

```
...

resources:
   requests:
      cpu: 4000m
      memory: 8192Mi
      qat.intel.com/generic: '1'
      intel.com/intel_sriov: '1'
   limits:
      cpu: 4000m
      memory: 8192Mi
      qat.intel.com/generic: '1'
      intel.com/intel_sriov: '1'

...
```