KEP: MultiNetwork podNetwork object - phase I

Goals

Introduce API object to kubernetes describing networks a Pod can attach to. Evolve current Kubernetes networking model to support multiple networks, defining the new model in a backwards compatible way. The current kubernetes networking would be represented by a default object.

Define “reference implementation” that can be used in Kubernetes CI.

Terminology

Cluster Default PodNetwork - This is the initial cluster-wide PodNetwork provided during cluster creation that is available to the Pod when no additional networking configuration is provided in Pod spec.
Primary PodNetwork - This is the PodNetwork inside the Pod whose interface is used for the default gateway (0.0.0.0) on the default VRF.
KCM - kube-controller-manager is a controller containing most of core kubernetes controllers

Background

Network

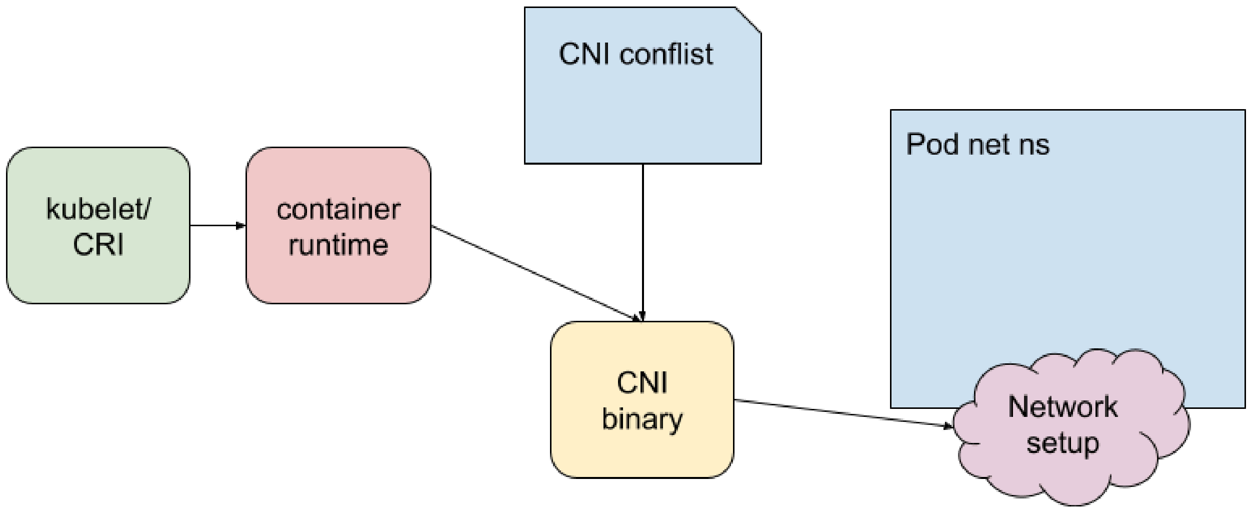
Network is a very overloaded term, and for many might have different meaning: it might be represented as a VLAN, or a specific interface in a Node, or identified as a unique IP subnet (a CIDR). In this design we do not want to limit to one definition, and ensure that we are flexible enough to satisfy anyone's definition.

CNI usage models

For the purpose of this design we want to call out the 2 models on how CNI API is used by various implementations.

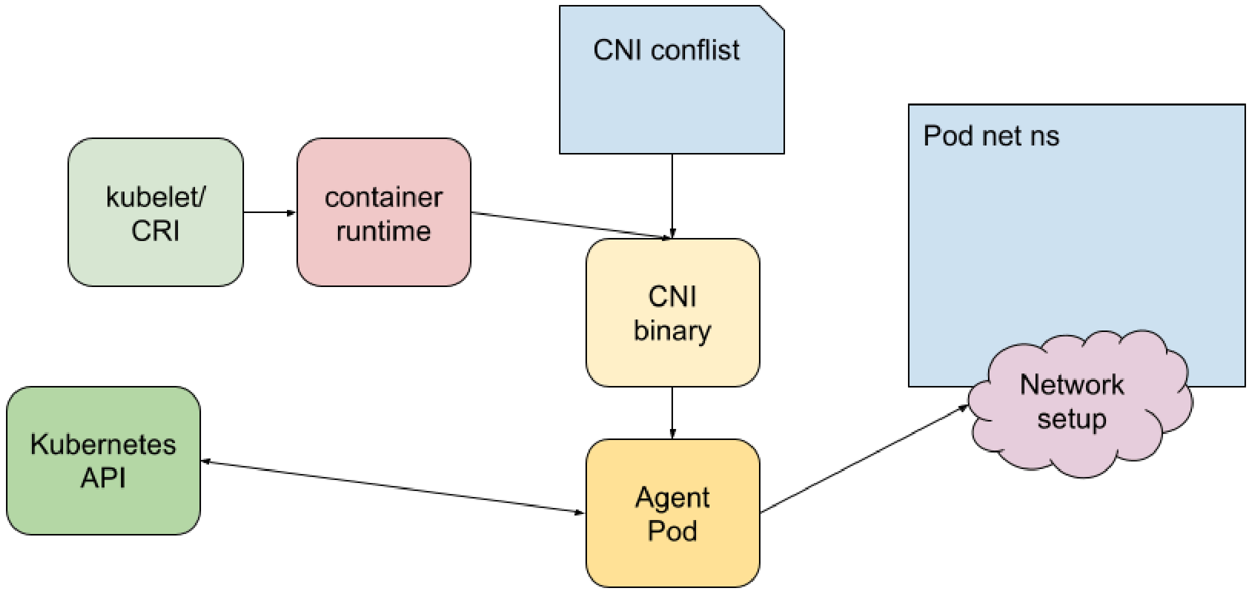
Standalone

This is the simplest and direct mode. Here we leverage in full the CNI API and all its configuration capabilities. All parameters required to set up a network inside a Pod network namespace are provided from the default kubelet API and the parameters of the local conflist file present in CNI configuration path (default /etc/cni/net.d/). The binary configures everything based on just these parameters and does not require any additional connection with Kubernetes API.



Agent-based

In this model, the CNI implementation is based on a fully fledged agent that usually runs as a hostNetwork Pod on each of the cluster Nodes. That agent has access to the Kubernetes API with RBAC defined permissions. The CNI binary is as simple as possible and communicates with the agent via local connections (e.g. socket file). Most of the logic is inside the agent. This agent does not rely on the CNI conflist, but gathers all the required data from Kubernetes API on its own (e.g. pulls the whole Pod object).



Overview

This design adds information to Pods to which “networks” (plural) it attaches to. Then the Pod scheduler will be able to understand which “network” is available and which is not in the cluster.

For that purpose this design introduces 2 new objects: `PodNetwork` and `PodNetworkAttachment`. `PodNetwork` is a core component that functions as a representation of a specific network in Kubernetes cluster, as well as a future handle for other objects to reference.
The `PodNetworkAttachment` is used as an additional Pod-level means to parameterize Pods’ network attachments.

Lastly we wish to introduce a new controller for these objects, to support the proposed life cycle.

Design Details

New Resources

PodNetwork

This KEP will add a `PodNetwork` object. This will be a core API object so that we can reference it in other Kubernetes core objects, like Pod. The core design principles for this object are:

1. PodNetwork’s form is generic and further specification is implementation-specific - is the network L2 or L3, or is it handled by CRI, CNI or a controller, should the network respect isolation or not, should all be the decision of implementation. We want to provide an API handle for the rest of the core or extended objects in Kubernetes.
2. PodNetworks may have overlapping subnets across them.
3. This enhancement is fully backward compatible and does not require any additional configuration from existing deployments to continue function.

PodNetwork object is described as follows:

```go
// +genclient
// +genclient:nonNamespaced
// +kubebuilder:object:root=true
// +kubebuilder:subresource:status
// +kubebuilder:resource:scope=Cluster

// PodNetwork represents a logical network in Kubernetes Cluster.
type PodNetwork struct {
	metav1.TypeMeta   `json:",inline"`
	metav1.ObjectMeta `json:"metadata,omitempty"`

```

```go
        Spec   PodNetworkSpec   `json:"spec"`
        Status PodNetworkStatus `json:"status,omitempty"`
}

// PodNetworkSpec contains the specifications for podNetwork object
type PodNetworkSpec struct {

        // Enabled is used to administratively enable/disable a PodNetwork.
        // When set to false, PodNetwork Ready condition will be set to False.
        // Defaults to True.
        //
        // +optional
        // +kubebuilder:default=true
        Enabled bool `json:"enabled,omitempty"`

        // ParametersRefs points to the vendor or implementation specific parameters
        // objects for the PodNetwork.
        //
        // +optional
        ParametersRefs []ParametersRef `json:"parametersRefs,omitempty"`

        // Provider specifies the provider implementing this PodNetwork.
        //
        // +kubebuilder:validation:MinLength=1
        // +kubebuilder:validation:MaxLength=253
        // +optional
        Provider string `json:"provider,omitempty"`
}

// ParametersRef points to a custom resource containing additional
// parameters for thePodNetwork.
type ParametersRef struct {
        // Group is the API group of k8s resource e.g. k8s.cni.cncf.io
        Group string `json:"group"`

        // Kind is the API name of k8s resource e.g. network-attachment-definitions
        Kind string `json:"kind"`

        // Name of the resource.
        Name string `json:"name"`

        // Namespace of the resource.
        // +optional
        Namespace string `json:"namespace,omitempty"`
}

// PodNetworkStatus contains the status information related to the PodNetwork.
```

```
type PodNetworkStatus struct{
        // Conditions describe the current conditions of the PodNetwork.
        //
        // Known condition types are:
        // * "Ready"
        // * "ParamsReady"
        //
        // +optional
        // +listType=map
        // +listMapKey=type
        // +kubebuilder:validation:MaxItems=5
        Conditions []metav1.Condition `json:"conditions,omitempty"`
}
```

Generic example:

```
apiVersion: v1
kind: PodNetwork
metadata:
  name: dataplane
spec:
  enabled: true
  provider: "foo.io/bar"
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: "2022-11-17T18:38:01Z"
    status: "True"
    type: Ready
```

Example with implementation-specific parameters:

```
apiVersion: v1
kind: PodNetwork
metadata:
  name: dataplane
spec:
  enabled: true
  provider: "k8s.cni.cncf.io/multus"
  parametersRefs:
  - group: k8s.cni.cncf.io
    kind: network-attachment-definitions
    name: parametersA
    namespace: default
  - group: k8s.cni.cncf.io
    kind: network-attachment-definitions
    name: complementaryParametersB
    namespace: default
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: "2022-11-17T18:38:01Z"
    status: "True"
    type: Ready
  - lastProbeTime: null
    lastTransitionTime: "2022-11-17T18:38:01Z"
    status: "True"
    type: ParamsReady
```

Status Conditions

The PodNetwork object will use following conditions:

- **Ready** - indicates that the PodNetwork object is correct (validated) and all other conditions are set to "true". This condition will switch back to "false" if ParamsReady condition is "false". Pods cannot be attached to a PodNetwork that is not Ready. This condition does not indicate readiness of specific PodNetwork on a per Node-basis. Following are the error reasons for this condition:

Reason name	Description
ParamsNotReady	The ParamsReady condition is not present or has "false" value. This can only happen when the "parametersRefs" field has a value.
AdministrativelyDisabled	The PodNetwork's Enabled field is set to false.

- **ParamsReady** - indicates that the objects specified in the "parametersRefs" field are ready for use. The implementation (effectively the owner of the specified objects) is responsible for setting this condition to "true" after having performed whatever checks the implementation requires (such as validating the CRs and checking that the network they reference is ready). The "Ready" condition is dependent on the value of this condition when the "parametersRefs" field is not empty. The available "reasons" for this condition are implementation specific. When multiple references are provided in the "parametersRefs" field, it is implementation responsibility to provide accurate status for all the listed objects using this one condition.

The conditions life-cycle will be handled by the PodNetwork controller described below.

Provider

The provider field gives the ability to uniquely identify what implementation (provider) is going to handle specific instances of PodNetwork objects. The value has to be in the form of a url. It is

the implementer's decision on how this field is going to be leveraged. They will decide how they behave when this field is empty and what specific value they are going to respect. This will dictate if a specific implementation can co-exist with other ones in the same cluster.
We have considered using classes (similar to GatewayClass etc.), but we do not expect any extra action for a PodNetwork to take place for specific implementation. PodNetwork already points to a custom resource, which implementation can hook on for any specific extra configuration. At this point of time we consider using classes as overkill, and in future if such need arises, we can reuse the provider field for that purpose.

Enabled

The Enabled field is created to allow proper migration from an existing PodNetwork. When set to `false` no new Pods can be attached to that PodNetwork.

IPAM

In this design we will not provide any specification for IPAM handling for PodNetworks. This will be specified in the following phases.

InUse state

The PodNetwork object can be referenced by at least one Pod or PodNetworkAttachment. When this is the case, the PodNetwork cannot be deleted. This will be maintained by the PodNetwork Controller and enforced via a finalizer.
When identifying Pods using PodNetwork for this purpose, the controller will filter out Pods that are in `Succeeded` or `Failed` state.

Mutability

The PodNetwork object will be immutable, except for the `Enabled` field.

Lifecycle

A PodNetwork will be in following phases:

1. Created - when the user just created the object and it does not have any conditions.
2. NotReady - when PodNetwork's Ready condition is `false`. Pods can reference such a PodNetwork, but will be in `Pending` state until the PodNetwork becomes Ready.
3. Ready - when validation of the PodNetwork succeeded and Ready condition is set to `true.` Here Pods can start attaching to a PodNetwork.
4. InUse - when there is a Pod or PodNetworkAttachment that references a given PodNetwork. PodNetwork deletion is blocked by a finalizer when InUse.
5. Disabled - when the user sets the Enabled field to `false`. We will mark such PodNetwork NotReady, and no new Pods will be able to attach to such PodNetwork.

Validations

We will introduce following validations for this object:

- Prevent mutation

- Ensure provider is a string
- Ensure listed parametersRef object fields are strings

This validation will be performed in the API server.

PodNetworkAttachment

The other object this KEP will add is `PodNetworkAttachment` object. This object will give providers the ability of a more detailed or per-Pod oriented configuration of the Pod attachment. In comparison, the PodNetwork object is a global representation of the network, and `PodNetworkAttachment` provides optional, more detailed configuration on a per-Pod level. PodNetworkAttachment object will have 2 functions:

- Provides ability to configure individual Interface of a Pod and still reference same PodNetwork
- Can contain status for an individual interface of a Pod

This is a namespaced object. It is described as follows:

```
// +genclient
// +genclient:Namespaced
// +kubebuilder:object:root=true
// +kubebuilder:subresource:status
// +kubebuilder:resource:scope=Namespaced

// PodNetworkAttachment provides optional pod-level configuration of PodNetwork.
type PodNetworkAttachment struct {
        metav1.TypeMeta   `json:",inline"`
        metav1.ObjectMeta `json:"metadata,omitempty"`

        Spec   PodNetworkAttachmentSpec   `json:"spec,omitempty"`
        Status PodNetworkAttachmentStatus `json:"status,omitempty"`
}

// PodNetworkAttachmentSpec is the specification for the PodNetworkAttachment resource.
type PodNetworkAttachmentSpec struct {
        // PodNetworkName refers to a PodNetwork object that this PodNetworkAttachment is
        // connected to.
        //
        // +required
        PodNetworkName string `json:"podNetworkName"`

        // ParametersRefs points to the vendor or implementation specific parameters
        // object for the PodNetworkAttachment.
        //
        // +optional
        ParametersRefs []ParametersRef `json:"parametersRefs,omitempty"`
}
```

```
// PodNetworkAttachmentStatus is the status for the PodNetworkAttachment resource.
type PodNetworkAttachmentStatus struct {
        // Conditions describe the current conditions of the PodNetworkAttachment.
        //
        // Known condition types are:
        // * "Ready"
        // * "ParamsReady"
        //
        // +optional
        // +listType=map
        // +listMapKey=type
        // +kubebuilder:validation:MaxItems=5
        Conditions []metav1.Condition `json:"conditions,omitempty"`
}
```

Example:

```
apiVersion: v1
kind: PodNetworkAttachment
metadata:
  name: pod1
  namespace: default
spec:
  podNetworkName: "dataplane"
  parametersRefs:
  - group: k8s.cni.cncf.io
    kind: podParams
    name: parametersA
    namespace: default
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: "2022-11-17T18:38:01Z"
    status: "True"
    type: Ready
  - lastProbeTime: null
    lastTransitionTime: "2022-11-17T18:38:01Z"
    status: "True"
    type: ParamsReady
```

Status Conditions

The PodNetworkAttachment will follow a similar life cycle as the PodNetwork object. One change will be a new error reason for Ready condition:

- **Ready** - indicates that the PodNetworkAttachment object is correct (validated) and ParamsReady condition is set to “true”, including the referenced PodNetwork’s Ready condition. This condition will switch back to “false” if any of the above conditions change to “false”. Pods cannot be attached to a PodNetworkAttachment that is not Ready. This condition does not indicate readiness of specific PodNetworkAttachment on a specific Node. Following are the error reasons for this condition:

Reason name	Description
ParamsNotReady	The ParamsReady condition is not present or has “false” value. This error can only happen when the “parametersRefs” field has a value.
PodNetworkNotReady	The referenced PodNetwork object Ready condition has “false” value.

The conditions life-cycle will be handled by the PodNetwork controller described below.

InUse indicator

The PodNetworkAttachment object can be referenced by at least one Pod. When this is the case, the PodNetworkAttachment cannot be deleted. This will be maintained by the PodNetwork Controller and enforced via a finalizer.
When identifying Pods using PodNetworkAttachment for this purpose, the controller will filter out Pods that are in `Succeeded` or `Failed` state.

Mutability

The PodNetworkAttachment object will be immutable. The API server will provide the admission control for that.

Lifecycle

The PodNetworkAttachment will be in following phases:
1. Created - when the user just created the object and it does not have any conditions.
2. NotReady - when PodNetworkAttachment’s Ready condition is `false`. Pods can reference such PodNetworkAttachment, but will be in `Pending` state until the PodNetwork becomes Ready.
3. Ready - when validation of the PodNetworkAttachment succeeded and Ready condition is set to `true`, here Pod can start attaching to a PodNetworkAttachment.
4. InUse - when there is a Pod that references a given PodNetworkAttachment. PodNetworkAttachment deletion is blocked by a finalizer when InUse.

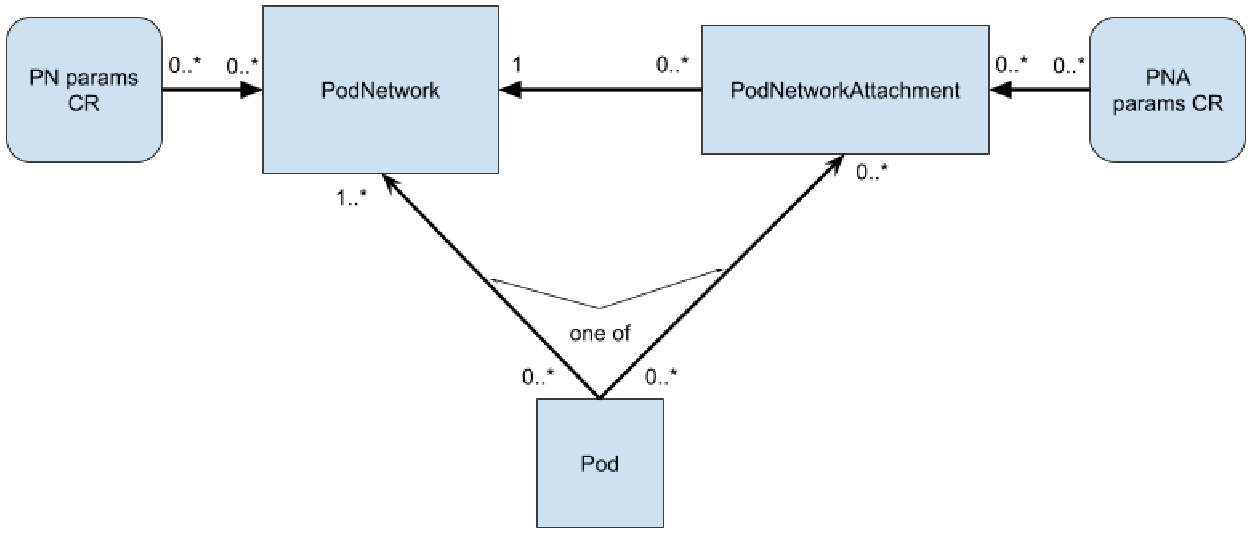
Validations

We will introduce following validations for this object:
- Prevent mutation
- Ensure podNetworkName is a string
- Ensure listed parametersRef object fields are strings

This validation will be performed in the API server.

Resources relations

The relation of the new objects to Pod and between each other is described in this diagram:



The arrows define which object references what object. Additionally:

- `PodNetworkAttachment` must reference exactly 1 `PodNetwork`
- `PodNetwork` can be referenced by multiple `PodNetworkAttachments`
- Pod can reference multiple `PodNetworks` or `PodNetworkAttachments`
- Pod has to reference at least 1 `PodNetwork` or `PodNetworkAttachment` (for backward compatibility, when nothing specified, “default” PodNetwork is auto-populated, see details below)
- For specific `PodNetwork` a Pod can either reference that `PodNetwork` or `PodNetworkAttachment` (referencing that `PodNetwork`), but not both at the same time
- `PodNetwork` can be referenced by multiple Pods
- `PodNetworkAttachment` can be referenced by multiple Pods
- `PodNetwork` can reference none, one or multiple parameters CRs
- `PodNetworkAttachment` can reference none, one or multiple parameters CRs
- Specific parameter CR can be referenced by one or more `PodNetwork` or `PodNetworkAttachment`, this is implementation specific decision

Default PodNetwork

We will introduce a “default” PodNetwork. This will be the Cluster Default PodNetwork. This PodNetwork will represent today's kubernetes networking done for Pods. This PodNetwork is characterized as:

- Will always be created during cluster creation (similarly to “default” kubernetes Namespace)
- All Pods not referencing any PodNetwork will connect to the “default” PodNetwork
- This PodNetwork will be named “default”

- The “default” PodNetwork must be available on all nodes
- Until it is created, all kubelets will report “Default PodNetwork not found” for their respective Nodes.

Availability

Considering “default” PodNetwork is critical for cluster functionality, we will provide special handling for it when it is being deleted. On deletion events, we will recreate it, so that it never has the deletionTimestamp field set. This is going to be handled by the API server.

In case “default” PodNetwork references any additional params in ParametersRef field, the implementer is responsible for those objects availability. “default” PodNetwork will become not Ready if ParamsReady condition is “false”.

Automatic creation

The PodNetwork controller will automatically create the “default” PodNetwork. The values set in it will be determined by the arguments passed to KCM.

Example:

```
apiVersion: v1
kind: PodNetwork
metadata:
  name: default
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: "2022-11-17T18:38:01Z"
    status: "True"
    type: Ready
```

Manual creation

We will introduce a new flag to KCM named `disable-default-podnetwork-creation` that will disable the automatic creation of the “default” PodNetwork. When specified, the Cluster Operator will be required to create the “default” PodNetwork. Until it is created, all kubelets will report “Default PodNetwork not found” for their respective Nodes.

Network Migration

This Phase will not implement the “default” PodNetwork migration procedure. It will arrive in a later time with a next phase. Following is the initial idea how it could work.
To change the configuration of Default PodNetwork, we will expose a new field in the Node spec object called `overrideDefaultPodNetwork`. This field will allow you to change what is the default PodNetwork on a per-Node basis. This field will be mutable. When kubelet is going to

report the presence of the Default PodNetwork, it will first look at this field, and then fallback to the “default” name.
For the migration process, the installer will have to set that field to the new PodNetwork, and when the “default” PodNetwork is no longer “InUse”, it can be deleted and replaced with a new version. At this point the installer will have to clear up the Node’s `overrideDefaultPodNetwork` field.

PodNetwork Controller

This KEP will introduce a new controller in KCM. Its main function will be:

- Handle PodNetwork and PodNetworkAttachment conditions
- Handle PodNetwork and PodNetworkAttachment finalizer to prevent deletion when InUse
- Automatically create the “default” PodNetwork

Feature gate

This feature will introduce a new feature gate to the `--feature-gates` argument and will be named “MultiNetwork”. All changes proposed in this design will be behind this gate.

Attaching PodNetwork to a Pod

We will extend the Pod spec with a new field `Pod.PodSpec.Networks`. It will be a list allowing attaching PodNetworks to a Pod. This list is explicit, and only the listed PodNetworks will be attached to the Pod. When the `Networks` field is not specified, Cluster Default PodNetwork will be attached (and set in that field) to the Pod (to keep backward compatibility). Proposed changes are as follows:

```
// PodSpec is a description of a pod.
type PodSpec struct {
[...]
        // Networks is a list of PodNetworks that will be attached to the Pod.
        //
        // +kubebuilder:default=[{podNetworkName: “default”}]
        // +optional
        Networks []Network `json:"networks,omitempty"`
}

// Network defines what PodNetwork to attach to the Pod.
type Network struct {
        // PodNetworkName is name of PodNetwork to attach
        // Only one of: [PodNetworkName, PodNetworkAttachmentName] can be set
        //
        // +optional
        PodNetworkName string `json:"podNetworkName,omitempty"`
        // PodNetworkAttachmentName is name of PodNetwork to attach
        // Only one of: [PodNetworkName, PodNetworkAttachmentName] can be set
        //
        // +optional
        PodNetworkAttachmentName string `json:"podNetworkAttachmentName,omitempty"`

        // InterfaceName is the network interface name inside the Pod for this attachment.
        // This field functionality is dependent on the implementation and its support
        // for it.
        // Examples: eth1 or net1
        //
        // +optional
        InterfaceName string `json:"interfaceName,omitempty"`

        // IsDefaultGW4 is a flag indicating this PodNetwork will hold the IPv4 Default
        // Gateway inside the Pod. Only one Network can have this flag set to True.
        // This field functionality is dependent on the implementation and its support
        // for it.
        //
        // +optional
        IsDefaultGW4 bool `json:"isDefaultGW4,omitempty"`

        // IsDefaultGW6 is a flag indicating this PodNetwork will hold the IPv6 Default
        // Gateway inside the Pod. Only one Network can have this flag set to True.
        // This field functionality is dependent on the implementation and its support
        // for it.
        //
        // +optional
        IsDefaultGW6 bool `json:"isDefaultGW6,omitempty"`
}
```

Example:

```
kind: Pod
metadata:
  name: pod1
  namespace: default
spec:
  containers:
…
  networks:
  - podNetworkName: dataplane
    interfaceName: net1  // optionally specify interfaceName
    isDefaultGW6: true
  - podNetworkName: default
    interfaceName: eth0
    isDefaultGW4: true
  - podNetworkAttachmentName: my-attach
    interfaceName: eth0
```

Static validations

We will perform static validation of the provided spec in the API Server, alongside the current Pod spec checks. These errors will be provided to the user immediately when they try to create Pod.
This will include:

- Ensure a single Pod references a given PodNetwork only 1 time
- IsDefaultGW4 and IsDefaultGW6 uniqueness for “true” value across multiple “Network” objects
- InterfaceName uniqueness across multiple “Network” objects
- InterfaceName naming constraints for Linux and Windows
- Ensure Network objects are not specified when hostNetwork field is set

The 1 PodNetwork per Pod restriction is coming from our current lack of details for future Multi-Networking requirements (e.g. Service support), and can be changed in future when we will discuss them.

Active validations

Beside the above, we will perform additional active validations, inside the scheduler, that require queries for other objects (e.g. PodNetwork). These errors will be presented as Pod Events, and the Pod will be kept in `Pending` state until the issues are resolved. We will do the following validation:

- Referenced PodNetwork or PodNetworkAttachment is present
- Referenced PodNetwork or PodNetworkAttachment is Ready
- Referenced PodNetworkAttachment, that is referencing a PodNetwork, follows the rule of: single Pod references a given PodNetwork only 1 time

Auto-population

When networks field is not set, and hostNetwork is not set, we will auto-populate this field with following values:

```
networks:
- podNetworkName: default
```

This is to ensure backward compatibility for clusters that will not use PodNetwork explicitly.

Status

All the IP addresses for attached PodNetworks will be present in the `Pod.PodStatus.PodIPs` list. To properly identify which IP belongs to what PodNetwork, we will expand the `PodIP` struct to include the name of PodNetwork the specific IP belongs to. This list will allow only 1 IP address per family (v4, v6) per PodNetwork.
The `Pod.PodStatus.PodIP` behavior will not change.
Proposed changes in **bold**:

```
// IP address information for entries in the (plural) PodIPs field.
// Each entry includes:
//
//      IP: An IP address allocated to the pod. ~~Routable at least within the cluster.~~
//      PodNetworkName: Name of the PodNetwork the IP belongs to.
//      InterfaceName: Name of the network interface inside the Pod.
type PodIP struct {
        // ip is an IP address (IPv4 or IPv6) assigned to the pod
        IP string `json:"ip,omitempty" protobuf:"bytes,1,opt,name=ip"`

        // PodNetworkName is name of the PodNetwork the IP belongs to
        //
        // +optional
        PodNetworkName string `json:"podNetwork"`

        // InterfaceName is name of the network interface used for this attachment
        //
        // +optional
        InterfaceName string `json:"interfaceName",omitempty`
}
```

Example:

```
kind: Pod
metadata:
  name: pod1
  namespace: default
spec:
…
  networks:
  - podNetwork: default
    interfaceName: eth0
    primary: true
  - podNetwork: dataplane1
    interfaceName: net1
  - podNetworkAttachmentName: my-interface-fast
    interfaceName: net2
```

```
status:
…
  podIP: 192.168.5.54
  podIPs:
  - ip: 192.168.5.54
    podNetwork: default
    interfaceName: eth0
  - ip: 10.0.0.20
    podNetwork: dataplane1
    interfaceName: net1
  - ip: 2011::233
    podNetwork: my-interface-fast
    interfaceName: net2
```

The above status is expected to be populated by kubelet, but this can only happen after CRI provides support for the new Pod API. Because of that, initially kubelet will behave as it does today, without updating the additional fields. Until CRI catches up, the PodNetwork providers will be able to update that field on their own.

DRA integration (alternative)

We have discussed, as an alternative model, usage of the DRA API, and concluded that using it will be less clear for the user, compared to the explicit PodNetwork model, proposed above.

API server changes

These are the changes for the API server covered by this design:

- Provide validation webhook for PodNetwork or PodNetworkAttachment objects
- Handle "default" PodNetwork deletion
- Extend static Pod spec validation

Scheduler changes

These are the changes we will do in Pod scheduler:

- Provide active Pod spec validation

When one of the multi-network validation fails, scheduler will follow the current "failure" path for Pod:

- set `PodScheduled` condition (of the Pod) to False with appropriate error message
- send Pod Event with same error message

Kubelet changes

We will introduce an additional check for kubelet readiness for networking. Today kubelet does this verification via CRI that checks CNI config presence. We will add a “default” PodNetwork presence check to this flow. Until such a PodNetwork is not created, kubelet will keep the Ready condition in “False” with “default PodNetwork not found in API” error message.

CRI changes

Considering the main input argument for kubelet when it interacts with CRI is the v1.Pod object, the above changes cover the kubelet-side part of providing the required data for multi-network. Next what is required are the changes to CRI API and CNI API which include

- Pod creation flow update in `RunPodSandbox`
- Pod status flow update in `PodSandboxStatus`
- CNI input and output values

Considering all above changes are in the direct scope of CRI, this KEP will not propose complete changes for them, and a separate KEP will be created to cover it.
Below are some suggestions on what the changes could look like.

Pod Creation

The Pod creation is handled by the `SyncPod` function, which calls `RunPodSandbox` (code) CRI API. The parameters for that function are defined by `PodSandboxConfig` (code). We propose change that API message in following way:

```
type PodSandboxConfig struct {
…
        // Optional configuration for PodNetworks.
        PodNetworks []*PodNetworkConfig `protobuf:"bytes,10,opt,name=podNetworks,proto3" json:"podNetworks,omitempty"`
}

// PodNetworkConfig specifies the PodNetwork configuration.
type PodNetworkConfig struct {
        // Name of the podNetwork.
        Name string `protobuf:"bytes,1,opt,name=name,proto3" json:"name,omitempty"`

        // Provider is the name of the implementer.
        Provider string `protobuf:"bytes,2,opt,name=provider,proto3" json:"provider,omitempty"`

        // InterfaceName name of the network interface inside the Pod namespace. Default: 0 (not specified).
        InterfaceName string `protobuf:"bytes,3,opt,name=interfaceName,proto3" json:"interfaceName,omitempty"`
        XXX_NoUnkeyedLiteral struct{} json:"-"
        XXX_sizecache        int32    json:"-"
}
```

Pod Status

This part is as well using the same `SyncPod` function, that gets all data from the `PodSandboxStatus` (code) CRI API. Internally there is `PodIP` structure to which we would like to add new fields `PodNetworkName` new `InterfaceName`:

```
// PodIP represents an ip of a Pod
type PodIP struct {
…
        // PodNetworkName name of the PodNetwork this IP belongs to
        PodNetworkName string   `protobuf:"bytes,2,opt,name=podNetworkName,proto3" json:"podNetworkName,omitempty"`

        // InterfaceName is name of the network interface inside Pod for this Interface/PodNetwork
        InterfaceName string   `protobuf:"bytes,3,opt,name=interfaceName,proto3" json:"interfaceName,omitempty"`
}
```

CNI API

Lastly, we would like for the CNI API to be able to handle Multi-Network for the agent-based CNI model (see above). It should have the ability to pass the list of PodNetworks requested by the Pod to the CNI binary, as well as be able to receive a list of IPs from the CNI with mapped PodNetworks.

Opens

1. How to handle RBAC access removal, post Pod creation, should we disconnect ?
2. How to support multiple IPs per family?
    a. This has been rejected as a requirements and is considered as a orthogonal effort for MN

Appendices

Proposal for use of DRA to implement PodNetworks

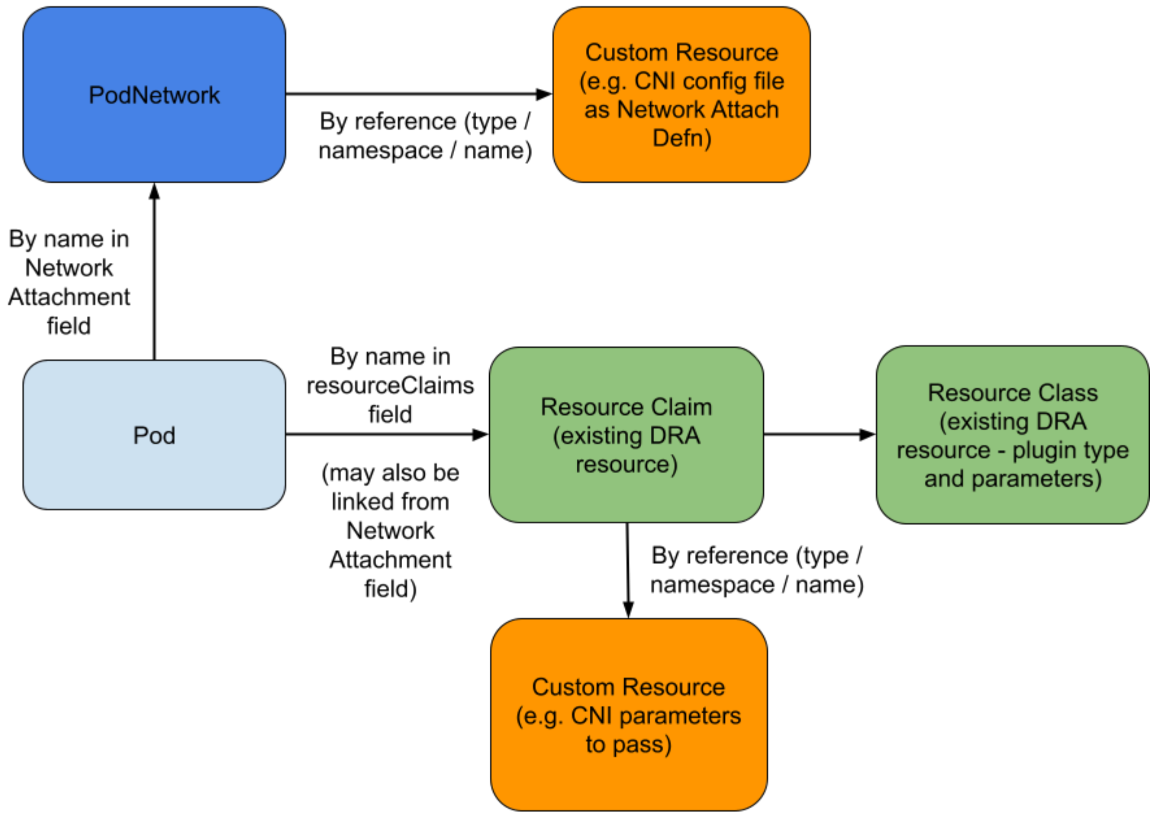
This section tries to lay out how DRA could be used to implement the pod network model. It describes how this works in a relatively simple case then expands to more complex cases. *This section is a draft: there are some open questions and various notes in purple show areas I am particularly keen to get feedback on.*

- TODO: clarify the explicit YAML details, including how ResourceClaims are linked to networks, and how Network Attachments and Interfaces work.

My view on the key points for discussion:

- DRA feels like a good solution, but we probably have to support two other models.
    - A minimal model suitable for the default network, which does not take any parameters and is available on all nodes (see below for why - mostly to do with bootstrapping the cluster).
    - Multus already exists, with plumbing including the use of device plugins and annotations. We must not break that. This almost comes for free - the default network can be “call the multus CNI”, and the annotations etc. pass through unchanged by this proposal.
- Pods in general need to be able to pass arbitrary parameters per network interface (as a reference to a CR). *This can be done via ResourceClaims; the exact details of how we create those claims is not clear yet, but we have several sensible similar options.*
- There are then a range of node level changes.
    - The kubelet is going to have to pass parameters to the CRI about networking. *The format of these is still unclear, and this is an area I do not know at all well - work in progress.*
    - The runtimes then must be updated to use those parameters and actually set up the parameters (out of scope for this specification, but work that needs to be done to have an implementation).
    - The CRI must return parameters to users. Some of these have to be returned to the kubelet over the CRI (to be written into the pod status - for example IP addresses), and some need to be passed to processes in the pod itself (say so that a container can do some device configuration, something multus would support using annotations and the downward API, but which might not be appropriate here).

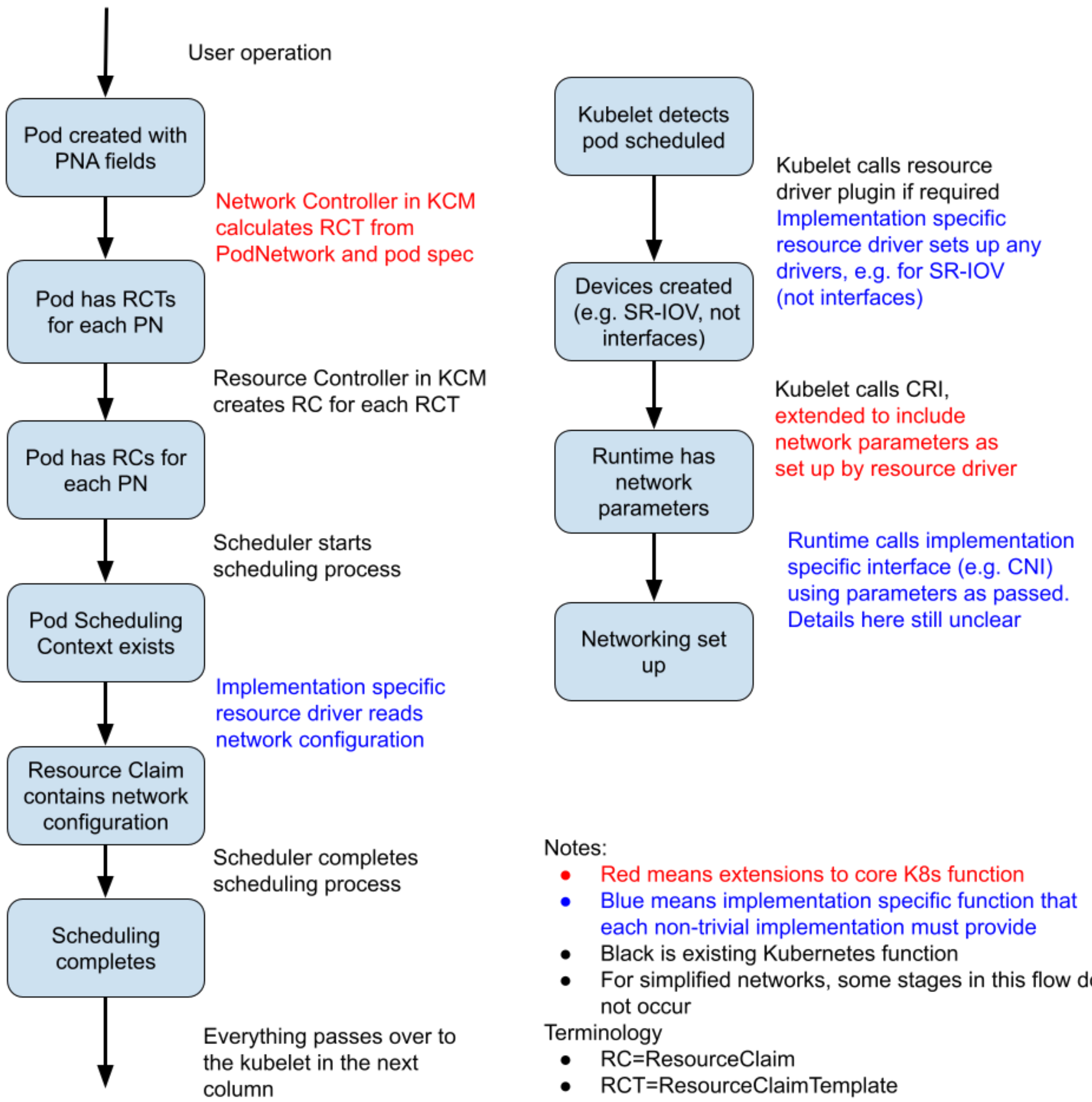
I’ll start with a picture of how the various resources link together.


PodNetwork
By reference (type / namespace / name)
Custom Resource (e.g. CNI config file as Network Attach Defn)
By name in Network Attachment field
Pod
By name in resourceClaims field
(may also be linked from Network Attachment field)
Resource Claim (existing DRA resource)
Resource Class (existing DRA resource - plugin type and parameters)
By reference (type / namespace / name)
Custom Resource (e.g. CNI parameters to pass)

In this picture:

- The new PodNetwork resource is in dark blue. This is the only new resource type, although the Pod is changed. DRA resources are shown in green, and custom resources in orange.
- The custom resources are all optional.
- I've shown pods as linking to explicit ResourceClaims. However, these are not reusable by large numbers of pods; in practice this can be resolved using ResourceClaimTemplates which lead to ResourceClaims being automatically created, but it would just make the discussion more complex so I am going to ignore this here.
- I've shown the CRs as CNI resources, as that is the simplest and most likely example, but that is not mandatory. Also note that there are parameters that are not in the CRs (such as interface name) that may be explicitly in the Pod Spec.
- Some open questions.
    - Where does the NetworkInterface resource exist?
    - There is some linkage from the Pod NetworkAttachment to the ResourceClaim; details of this are still TBD.

And here is a picture of the flow when a pod is created (ignoring simplified networks that are discussed below).


User operation
Pod created with PNA fields
Network Controller in KCM calculates RCT from PodNetwork and pod spec
Pod has RCTs for each PN
Resource Controller in KCM creates RC for each RCT
Pod has RCs for each PN
Scheduler starts scheduling process
Pod Scheduling Context exists
Implementation specific resource driver reads network configuration
Resource Claim contains network configuration
Scheduler completes scheduling process
Scheduling completes
Everything passes over to the kubelet in the next column
Kubelet detects pod scheduled
Kubelet calls resource driver plugin if required
Implementation specific resource driver sets up any drivers, e.g. for SR-IOV (not interfaces)
Devices created (e.g. SR-IOV, not interfaces)
Kubelet calls CRI,
extended to include network parameters as set up by resource driver
Runtime has network parameters
Runtime calls implementation specific interface (e.g. CNI) using parameters as passed. Details here still unclear
Networking set up

Notes:
- Red means extensions to core K8s function
- Blue means implementation specific function that each non-trivial implementation must provide
- Black is existing Kubernetes function
- For simplified networks, some stages in this flow do not occur

Terminology
- RC=ResourceClaim
- RCT=ResourceClaimTemplate
- PN=PodNetwork
- PNA=PodNetworkAttachment, with link to PN and optional parameters

DRA reminder

As a reminder of the key points of Dynamic Resource Allocation (as it stands today):

- The PodSpec is extended to have an optional ResourceClaims list. Each element in that list is a link to a ResourceClaim resource.
- A ResourceClaim is a core K8s resource that represents the claim for a particular type of resource, and it contains two important fields.
  - A link to a ResourceClass
  - An optional link to a CustomResource which can contain arbitrary content specific to this claim

Think of the ResourceClass as a resource that specifies the type of resource (e.g. “a GPU of model X”), and the linked CR as containing arbitrary parameters for that resource (such as “with capacity Y”). Resource claims are normally specific to pods (there are exceptions that don’t matter here because they are limited to very small numbers of pods sharing a claim).

- Rather than explicitly creating a ResourceClaim resource for every pod, a pod may reference a ResourceClaimTemplate. This leads to creation of a ResourceClaim for the pod (and its deletion when the pod is deleted). A ResourceClaimTemplate has much the same fields in it as a ResourceClaim.
- The ResourceClass resource tells Kubernetes how to handle provisioning the resource. It contains the following important fields.
    - A driver name, that is just an identifier used elsewhere
    - An optional link to a CustomResource with driver specific parameters
    - An optional node selector, indicating which nodes can satisfy the claim (missing implies any node - this is a simplified scheduling option as an alternative to the more complex model below)

When writing a DRA driver, there are two key pieces required.

- A controller which must:
    - Interact with scheduling, updating PodSchedulingContexts.
    - Perform allocation for ResourceClaims as appropriate (doing any reservation of resources for the claim - which might happen when the claim is created or when a pod first uses it, depending on the parameters in the ResourceClaim).
- A DRA plugin that is installed on each node, and communicates with the kubelet by a gRPC interface on a UNIX socket. The kubelet passes information to that plugin that it can use to instantiate the device and link it to the containers.

The core Kubernetes use of DRA is then as follows.

- When scheduling pods, the scheduler detects that there are ResourceClaims in the PodSpec. The scheduler and the DRA controllers then use PodSchedulingContext resources and ResourceClaim status fields (via the API server) to go through a flow where the pod ends up scheduled on a node that can satisfy the claims, and the claims have the appropriate devices associated with them.
- Once the pod reaches a node, the kubelet makes a gRPC call to the DRA plugin to let it do what it needs to (such as configure the device in question and connect it to the pod).

In comparison with Device Plugins (which are routinely used now):

- DRA allows the passing of arbitrary configuration parameters to a device.
- Device Plugins support integer counts of resources; DRA supports much more complex scheduling capabilities using controllers, and closes some timing windows in the scheduling flow.

Option 1: Using DRA to do everything

In this simplest case the proposed model works as follows. *This does not take account of more complex scenarios; think of "just call the CNI on a node that supports that CNI". That will come later.*

- There are two new components.
  - There is a new PodNetwork controller (not strictly a controller, but linked to the Pod Controller) whose job it is to manage PodNetworkAttachments in pods. This is a change to core Kubernetes.
  - There is a new networking DRA driver created by this group and installed where multi-network with more than simplified networks is required. This driver has both a controller and a plugin to handle the PodNetwork model. It is not a change to core Kubernetes, but an implementation of the existing DRA interfaces.
- There are a range of PodNetwork resources, each of which has:
  - an optional linked CR with network specific configuration (for CNI, this would be the NetworkAttachmentDefinition, i.e. CNI config file);
  - optional links to some ResourceClaimTemplates, one of which is marked as the default. Their use is described just below.
- When users create a pod, the pod has a PodNetworkAttachment array field indicating which PodNetworks the pod should be attached to. (If there is no such array or the array is empty, then the pod just uses the default network - more about that in the next section.)
- After the pod is created in the API server, the new PodNetwork controller creates a ResourceClaim for each PodNetworkAttachment field in the pod. The parameters for this ResourceClaim come from a ResourceClaimTemplate from one of the following sources.
  - The PodNetworkAttachment may contain no indication of network config. In this case, the default ResourceClaimTemplate from the PodNetwork is used.
  - The PodNetworkAttachment may contain the name of a ResourceClaimTemplate linked to from the PodNetwork (as a field "NetworkConfigurationName" or something similar), in which case the ResourceClaimTemplate with that name from the PodNetwork is used.
  - Finally, the PodNetworkAttachment may reference an explicit ResourceClaimTemplate or ResourceClaim created by the user for this pod.

  In this way, when attaching a pod to a network, the user can accept the default configuration, specify a predefined configuration, or create their own configuration. Since the configuration contains references to CRs, this can contain any desired networking configuration.
- When pod scheduling starts, the controller of the networking DRA driver gets involved. (This is standard DRA behaviour, and requires writing a networking DRA driver as mentioned above but no changes to core Kubernetes.)
  - The controller updates the scheduling decision so that the pod is scheduled where the network is supported.
  - The "AllocationResult" field in the ResourceClaimStatus contains information to be passed to the plugin (see below). The controller updates that field with the

        content of the PodNetwork, the PodNetwork's linked CR, and the ResourceClaim's linked CR (if any).

- After scheduling, the pod is instantiated by the runtime. The usual DRA model means that the information about the network stored above is passed to the device plugin via a local gRPC interface. *There are some questions here - we have got the data to the node, but not into the CRI. One simple way to resolve this is for the same configuration data passed to the plugin just to be added to new CRI networking fields, but there is more detail to work through.*

There are a couple of problems with this model that need to be addressed (and are covered in the next paragraph).

- There is an ordering problem, where at least the default network must function before any pods are running (to avoid pod creation requiring a network controller that relies on pods having been created). I think this is a real problem that we need to fix, but I'm open to some DRA expert telling me I have misunderstood.
- For users who just do not want any extra network function or multiple networks, everything has got worse.
    - There is one ResourceClaim per pod being created, which is very visible to users and is going to provoke confusion.
    - The scheduling has become slower and more complex, which affects the mainline for users who do not want the function. I think this is a valid concern - comments welcome.

To resolve these problems, it is possible for a PodNetwork to opt for a simplified model (as specified in the PodNetwork resource). The default PodNetwork always opts for the simplified model. In that case:

- ResourceClaims are not created. Scheduling does not need to take account of the network (and so the network must exist on all nodes - which is already true for the default PodNetwork), and so is unchanged.
- There is no way to specify any parameters on the network.
- No information is passed via gRPC to any plugin. That means that for simplified model PodNetworks, we must somehow inform the runtime of the network names (but of nothing else, since there are no parameters).

This simplified model is a duplication of effort that I think we may just have to accept as necessary - feedback welcome.

Passing parameters (open questions)

- For a pod to pass specific parameters to the implementation, the description above allows for arbitrary parameters, but if there is a wish that (say) a PodNetworkAttachment specifies an arbitrary interface name then each pod will need to have a ResourceClaimTemplate created for it by the user, which is quite ugly. That probably implies that we want to have some way of passing some parameters that is lower effort for common parameters. There are a couple of options here which could help with this.

    - We could add some fields to the PodNetworkAttachment for common items such as “Interface name”.
    - We could let users create NetworkInterface resources that are linked to from the pod with the fields in them.
- Ultimately, outputs need to be passed back to the pod and the apiserver (which might in principle get different subsets of data). We currently don’t have a detailed model for this beyond the need to set up PodIPs with network names.

Alternative model with Device Plugins (dismissed)

An alternative approach would be that instead of automatically adding ResourceClaims, the control plane could automatically add resource requests based on device plugins, and we could create a networking device plugin. This is simpler than the DRA based model, but the Device Plugin gRPC API may not be rich enough for our needs. I don’t know enough about the API to definitely dismiss this, so feedback welcome.
Based on feedback so far, I think the Device Plugin model we should support is just “if you have multus, it will continue to work so go on using multus, and if you want a new network move to the new model with DRA”.

Option 2: Using DRA only for complex cases

In this model, the approach is rather different.

- DRA only plays a role where p I’m ods have requirements that are more complex than the usual defaults
- In addition to having (as today) ResourceClaim links in containers, these may also be explicitly specified in PodNetworkAttachments.
- When this occurs, the usual DRA flow occurs.
    - The DRA controller (which is in general network specific, but may have device specific behaviour too such as for setting up SR-IOV network devices) is responsible for any more complex scheduling capability required.
    - The DRA plugin is responsible for passing extra information to whatever implements the network (or is itself whatever implements the network).

Multus based model

Suppose a cluster is using multus for the network implementation. There are then several stages of migration possible.

- With a multus enabled cluster, upgrading Kubernetes (and the runtimes etc.) to a new version does not break anything - the default network is implemented using the multus CNI, which continues to read annotations etc.
    - If a pod is just to be on the default network, then it is set up by multus (which does not set up additional networks if there are no annotations).
    - If new networks are to be added and managed by multus, then this is perfectly possible.
- It would be possible to configure additional networks so that they also use multus. In this case, instead of multus creating the default CNI plugin and optionally other CNI plugins

based on annotations, multus would be called with a different default CNI. *I doubt there is a good use case for this, but it's possible.*

- If there is a wish to stop multus calling a CNI and have it called directly, this can be done by replacing the annotation parameters passed for multus by some CR linked to from the PodNetworkAttachment (and assumed to be passed to the runtime above). Then the CNI can be called as normal. The required changes for this are:
    - That is a significant change for the CRI / runtime (though something like this is assumed above).
    - This is a user visible change - users create ResourceClaims with linked CRs rather than annotations.
    - Without multus there is no way to write annotations back to the pod. However, the above discussion assumed that we'd come up with such a model anyway.

  The key point is that this can be done delegate by delegate; there is a gradual migration path.

PodNetwork example with multus SRIOV

Example for HW-base network:

```
apiVersion: v1
kind: PodNetwork
metadata:
  name: sriov
spec:
  ipam4: "kubernetes"
  provider: "k8s.cni.cncf.io/multus"
  parametersRefs:
  - group: k8s.cni.cncf.io
    kind: network-attachment-definitions
    name: sriov
    namespace: default
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: "2022-11-17T18:38:01Z"
    status: "True"
    type: Ready
  - lastProbeTime: null
    lastTransitionTime: "2022-11-17T18:38:01Z"
    status: "True"
    type: ParamsReady
---
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
   name: sriov
   namespace: default
   annotations:
     k8s.v1.cni.cncf.io/resourceName: my.co.io/device
 spec:
   config: '{
       "cniVersion": "0.3.0",
       "name": "mynetwork",
       "type": "sriov",
       "ipam": {
         "type": "whereabouts",
         "range": "21.0.108.0/21",
         "range_start": "21.0.111.16",
         "range_end": "21.0.111.18"
       }
     }'
 ---
 apiVersion: v1
 kind: Pod
 metadata:
   name: pod-sriov
   namespace: default
 spec:
  …
   podNetwork:
     -   sriov
   resources:
     limits:
       my.co.io/device: 1      ←– still done by mutating webhook
     requests:
       my.co.io/device: 1
  status:
  …
   podIP: 192.168.5.54
   podIPs:
   - ip: 192.168.5.54
     podNetwork: default
   - ip: 10.0.0.20
     podNetwork: sriov
```

IPAM future design considerations

PodNetwork object is described as follows:

```
// IPAMType defines source of Pods IPAM handling.
// +kubebuilder:validation:Enum=external;kubernetes;none
```

```go
type IPAMType string

const (
        // External uses external mechanisms to define IPAM configuration.
        // The implementation must return an IP for the attachment.
        External IPAMType = "external"

        // Kubernetes uses a built-in mechanism to configure IPAM.
        // Based on KCM NodeIPAM controller and ClusterCIDR.
        // The implementation must return an IP for the attachment.
        Kubernetes IPAMType = "kubernetes"

        // None option is used when no IP will be present and
        // reported on the attachment to this PodNetwork.
        NoneType IPAMType = "none"
)


// +genclient
// +genclient:nonNamespaced
// +kubebuilder:object:root=true
// +kubebuilder:subresource:status
// +kubebuilder:resource:scope=Cluster

// PodNetwork represents a logical network in Kubernetes Cluster.
type PodNetwork struct {
        metav1.TypeMeta   `json:",inline"`
        metav1.ObjectMeta `json:"metadata,omitempty"`

        Spec   PodNetworkSpec   `json:"spec"`
        Status PodNetworkStatus `json:"status,omitempty"`
}

// PodNetworkSpec contains the specifications for podNetwork object
type PodNetworkSpec struct {

        // IPAM4 defines what is handling v4 IPAM for Pods attaching to
        // this PodNetwork.
        // When not specified, IPv4 configuration is undefined and is not
        // expected nor reported on the attachment to this PodNetwork.
        // +optional
        IPAM4 *IPAMType `json:"ipam4,omitempty"`

        // IPAM6 defines what is handling v6 IPAM for Pods attaching to
        // this PodNetwork.
        // When not specified, IPv6 configuration is undefined and is not
        // expected nor reported on the attachment to this PodNetwork.
        // +optional
        IPAM6 *IPAMType `json:"ipam6,omitempty"`

…

        // Provider specifies the provider implementing this PodNetwork.
        // +optional
        Provider string `json:"provider,omitempty"`
}

// ParametersRef points to a custom resource containing additional
// parameters for thePodNetwork.
type ParametersRef struct {
        // Group is the API group of k8s resource e.g. k8s.cni.cncf.io
        Group string `json:"group"`

        // Kind is the API name of k8s resource e.g. network-attachment-definitions
        Kind string `json:"kind"`

        // Name of the resource.
        Name string `json:"name"`

        // Namespace of the resource.
        // +optional
        Namespace string `json:"namespace,omitempty"`
}
```

IPAM fields

The IPAM fields (v4 and v6), indicate a few things:

- Is an IP required from a specific attachment for a specific family?
- Is the Kubernetes built-in NodeIPAM used?
- Indirectly, if this is a single stack (v4 or v6) or dualstack attachment

The implementation for `Kubernetes` type IPAM for the non-Default PodNetworks will come as a separate KEP, expanding on the ClusterCIDR object (KEP currently being modified in PR).

The check if an IP address has been provided for specific attachment (based on this field), will be introduced in a later phase when CRIs implement support for multi-network.

One IP per family

Today a kubernetes Pod is represented by a single IP address per IP family (v4 and v6). We want to preserve the same behavior in PodNetwork where one attachment to a Pod from one PodNetwork can be represented by at most a single IP address per IP family. Additionally, PodNetworks will support networks that do not provide L3 connectivity, and in such cases will not provide any IP.

Mutability

The PodNetwork object will be immutable with a caveat for the IPAM4 and IPAM6 fields. These fields, if not specified, can be set (via update), but not changed when they already have a value.

The API server will provide the admission control for the above. In addition, implementers of PodNetwork may provide additional restrictions in their own admission controllers.

Validations

We will introduce following validations for this object:

- Prevent mutation (with caveat mentioned above)
- Ensure ipam4 and ipam6 fields use proper enumerate
- Ensure provider is a string
- Ensure listed parametersRef object fields are strings

Automatic creation

The PodNetwork controller will automatically create the “default” PodNetwork. The values set in it will be determined by the arguments passed to KCM.
Based on KCM options (link) we do the following:

- `allocate-node-cidrs` will control the value of IPAM4 and IPAM6 fields. When set “true” we will use "kubernetes" value, otherwise we will not set any value, and allow the implementation to update the “default” PodNetwork with correct values.
- `cluster-cidr` will control how each of IPAM4 and IPAM6 fields should be set. When the value is single stack we will set the respective IP family to "kubernetes" and the other one to “none”.
  When both IP families are set we will set both fields to "kubernetes".

Example for dualstack (`allocate-node-cidrs=true`, `cluster-cidr=”192.168.0.0/16,2001::/96”` doc):

```yaml
apiVersion: v1
kind: PodNetwork
metadata:
  name: default
spec:
  ipam4: "kubernetes"
  ipam6: "kubernetes"
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: "2022-11-17T18:38:01Z"
    status: "True"
    type: Ready
```

Example for single stack (IPv4) (`allocate-node-cidrs=true`, `cluster-cidr=”192.168.0.0/16”`) :

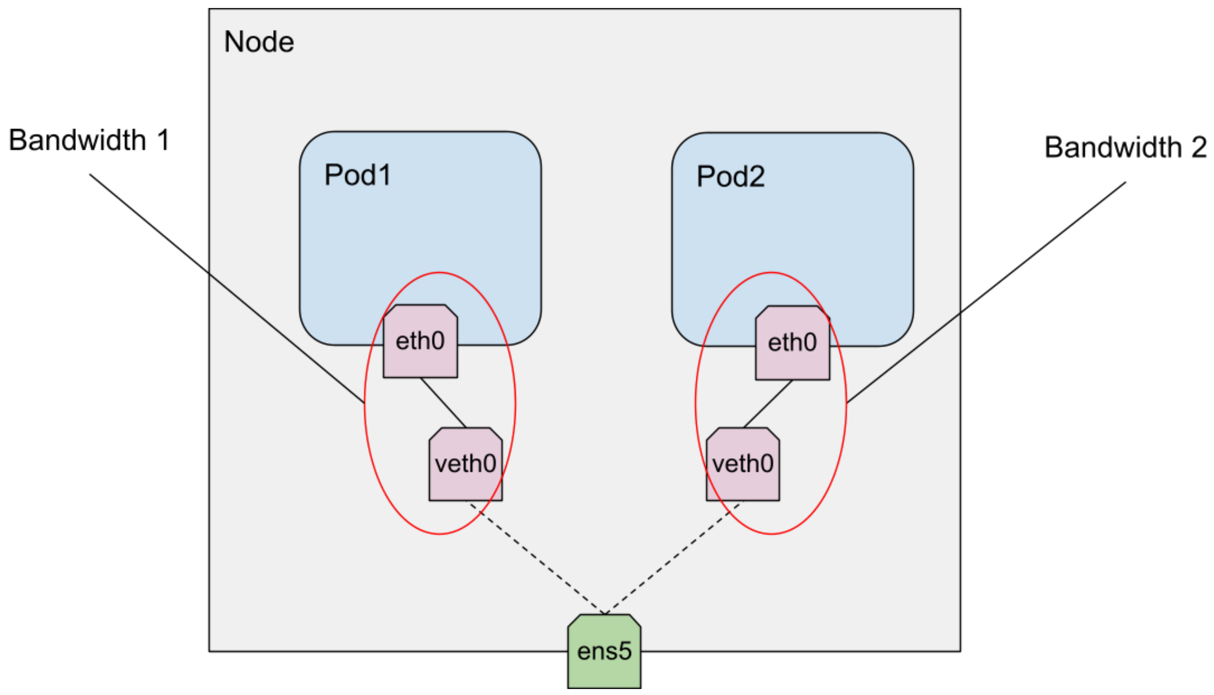
```
apiVersion: v1
kind: PodNetwork
metadata:
  name: default
spec:
  ipam4: "kubernetes"
  ipam6: "none"
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: "2022-11-17T18:38:01Z"
    status: "True"
    type: Ready
```

Example for no configuration (`allocate-node-cidrs=false`):

```
apiVersion: v1
kind: PodNetwork
metadata:
  name: default
spec: {}
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: "2022-11-17T18:38:01Z"
    status: "True"
    type: Ready
```

PodNetworkAttachment Use Case

As Cluster operator I wish to manage my cluster’s bandwidth usage on a per pod basis, simultaneously preserving all other Pod networking configuration the same across those Pods.


Node
Bandwidth 1
Pod1
Pod2
Bandwidth 2
eth0
eth0
veth0
veth0
ens5