KubeCon | CloudNativeCon

Europe 2020

Virtual

Building a Software-Defined-Network Using K8s API Machinery and Controllers

Matteo Olivi, Independent

References

Authors:

Matteo Olivi, recent Comp Eng MSc from the University of Bologna:
github: matte21
email: matteoolivi7@gmail.com
slack: matte21

Mike Spreitzer, principal RSM at IBM Research:
github: MikeSpreitzer
email: mspreitz@us.ibm.com
slack: mspreitz

Project: https://github.com/MikeSpreitzer/kube-examples/tree/add-kos/staging/kos

Why do this?

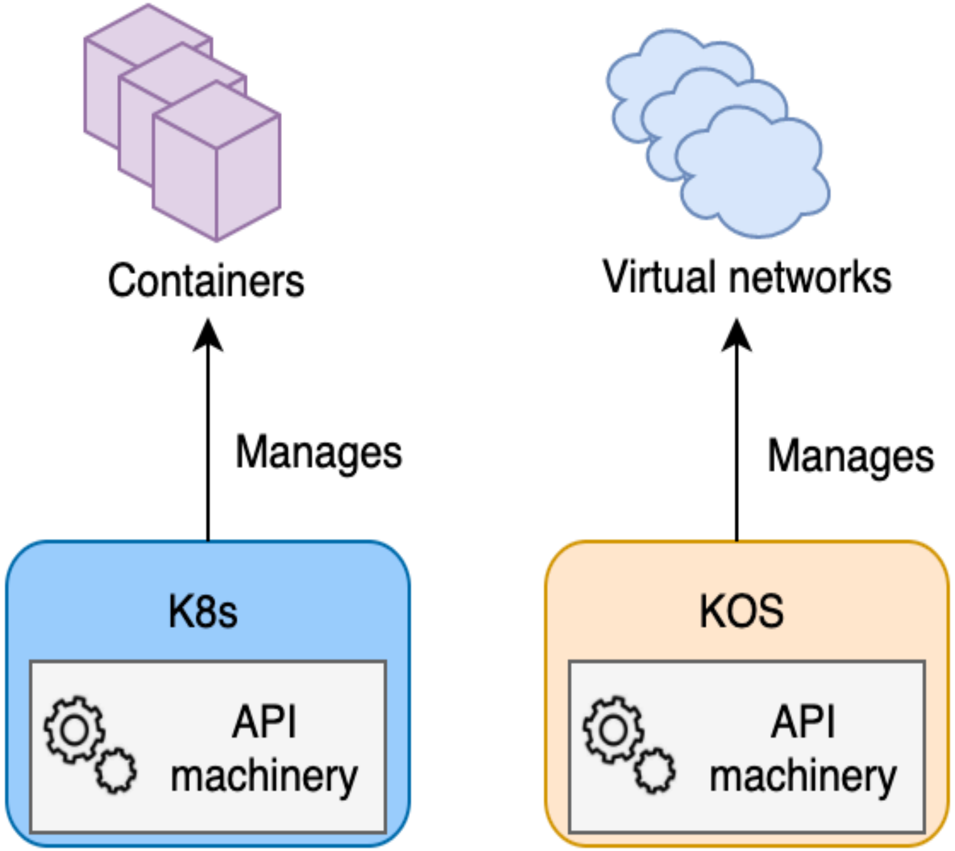
Test and demonstrate that **K8s controller pattern** and **API machinery** are general building blocks, not just for K8s.

We built **KOS** (K8s OvS SDN).

KOS manages VXLAN virtual networks.

NOT for production.

Runs only on Linux.



Agenda

1. VXLAN.
2. K8s controller pattern.
3. KOS API.
4. KOS architecture.
5. Interesting, general challenges.

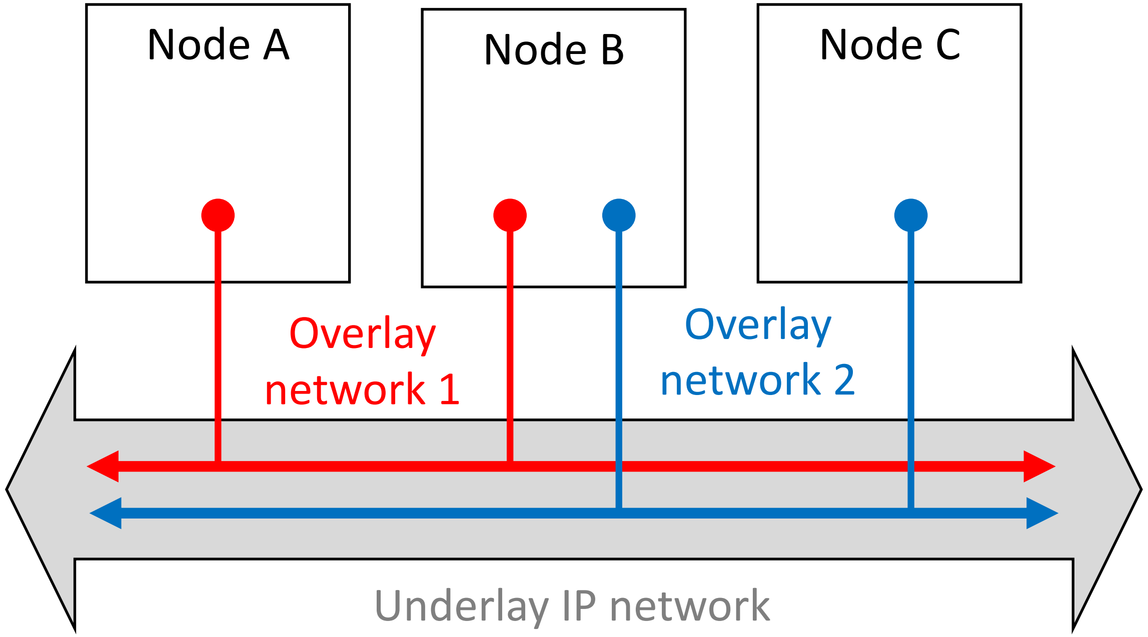
VXLAN

Goal: overlay virtualized Ethernet networks over IP underlay networks.

Features:

- Isolation and scalability.
- Tunneling-via-encapsulation protocol.

KOS dynamically configures VXLAN overlay networks.



K8s controller pattern

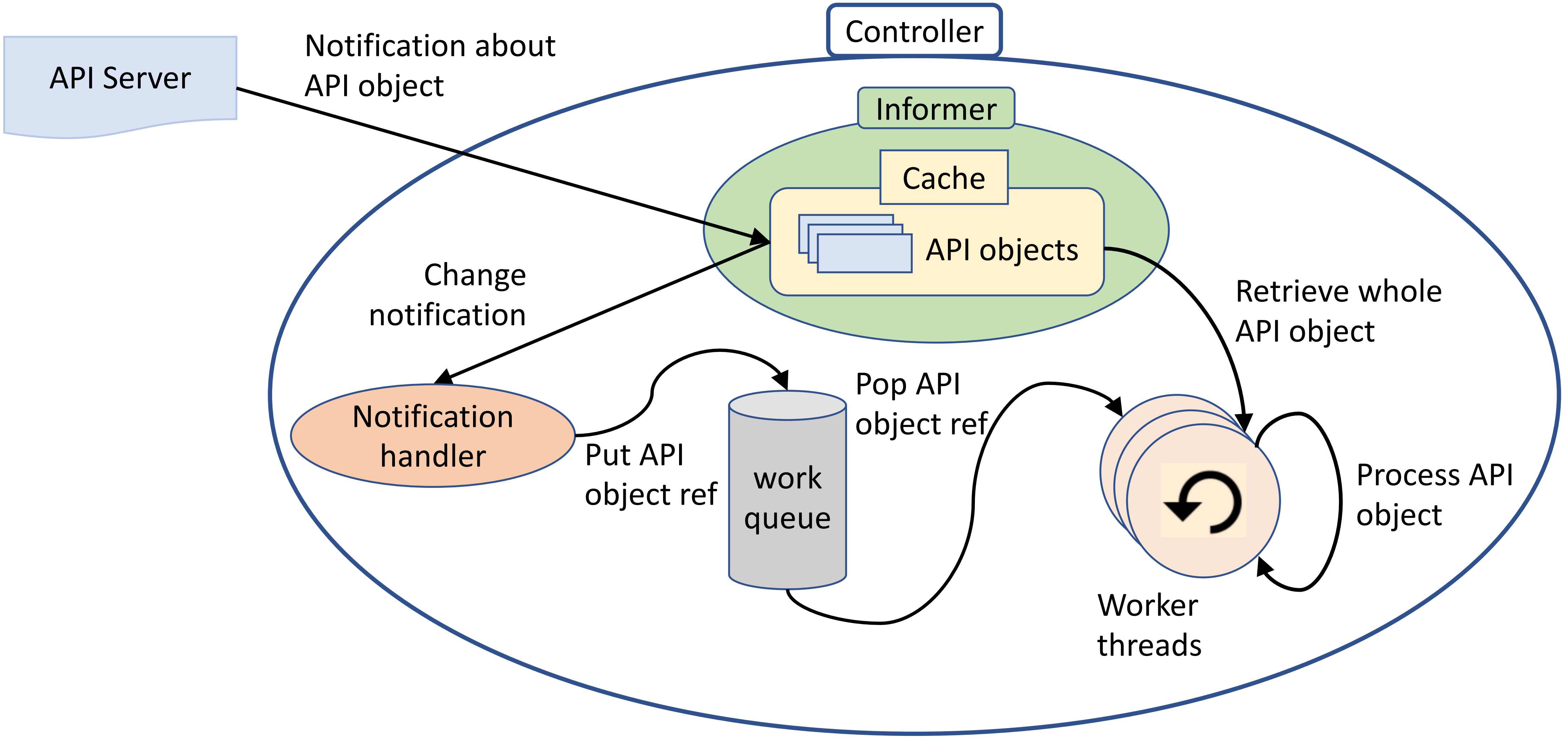
An API object has two sections:

- desired state (*spec*)
- observed state (*status*)

K8s control plane is a set of control loops (*controllers*) that:

- listen for API objects notifications (create/update/delete).
- modify “real world” to drive it towards API objects desired state.
- write back observed, real world state into API object *status* (with *MVCC*).

Anatomy of a K8s controller



KOS API

Like in K8s, users interact with KOS via CRUD operations on API objects.

KOS defines **three custom API object types**:

- **Subnet**: an IPv4 subnet in a VXLAN virtual network.
- **NetworkAttachment (NA)**: an interface on a VXLAN virtual network.
- **IPLock**: more details later.

Custom types are implemented in custom API servers.

API objects examples

```
apiVersion: network.example.com/v1alpha1
kind: Subnet
metadata:
  name: s1
  namespace: ns1
spec:
  vni: 1
  ipv4: 192.168.10.0/24
status:
  validated: true
```

```
apiVersion: network.example.com/v1alpha1
kind: NetworkAttachment
metadata:
  name: na1
  namespace: ns1
spec:
  subnet: s1
  node: node1
status:
  guestIP: 192.168.10.0
  guestMAC: 02:a8:0a:00:00:01
  addressVNI: 1
  hostIP: 10.190.65.131
```

NA implementation

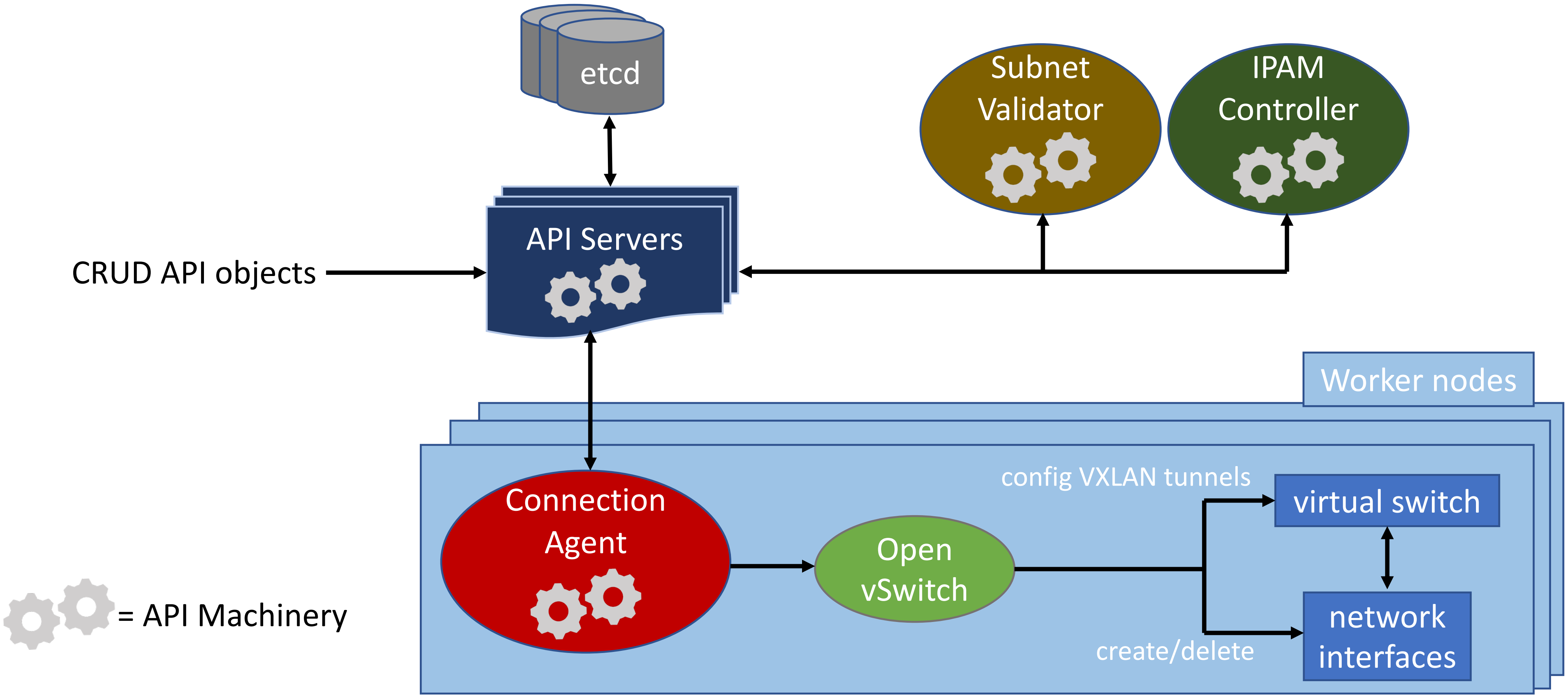
KOS implements a *NA* with:
1. A virtual IP address.
2. A virtual MAC address.
3. A Linux network interface on the underlay node of the *NA*.

Plus…

On every underlay node, there’s an OvS switch that supports VXLAN.

KOS:
1. connects the network interface to the switch.
2. sets up the switch to ENCAP/DECAP all traffic from/to the network interface.

KOS architecture

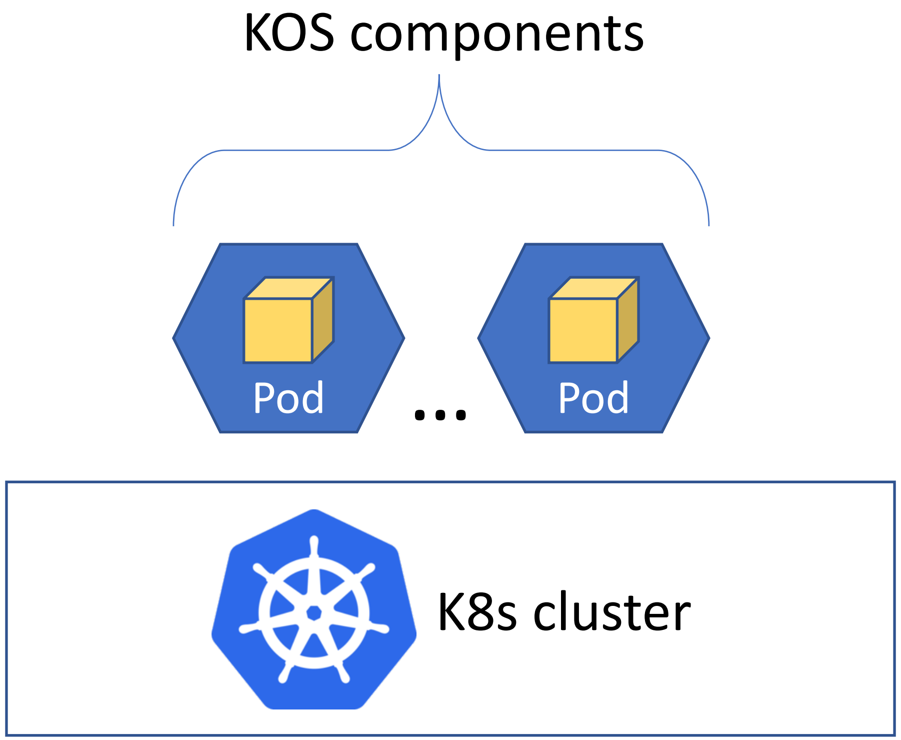


KOS Deployment

KOS runs as a K8s workload.

But…

could also be deployed on its own.



Multi-object invariants pt. 1

For Subnets with same *vni*:

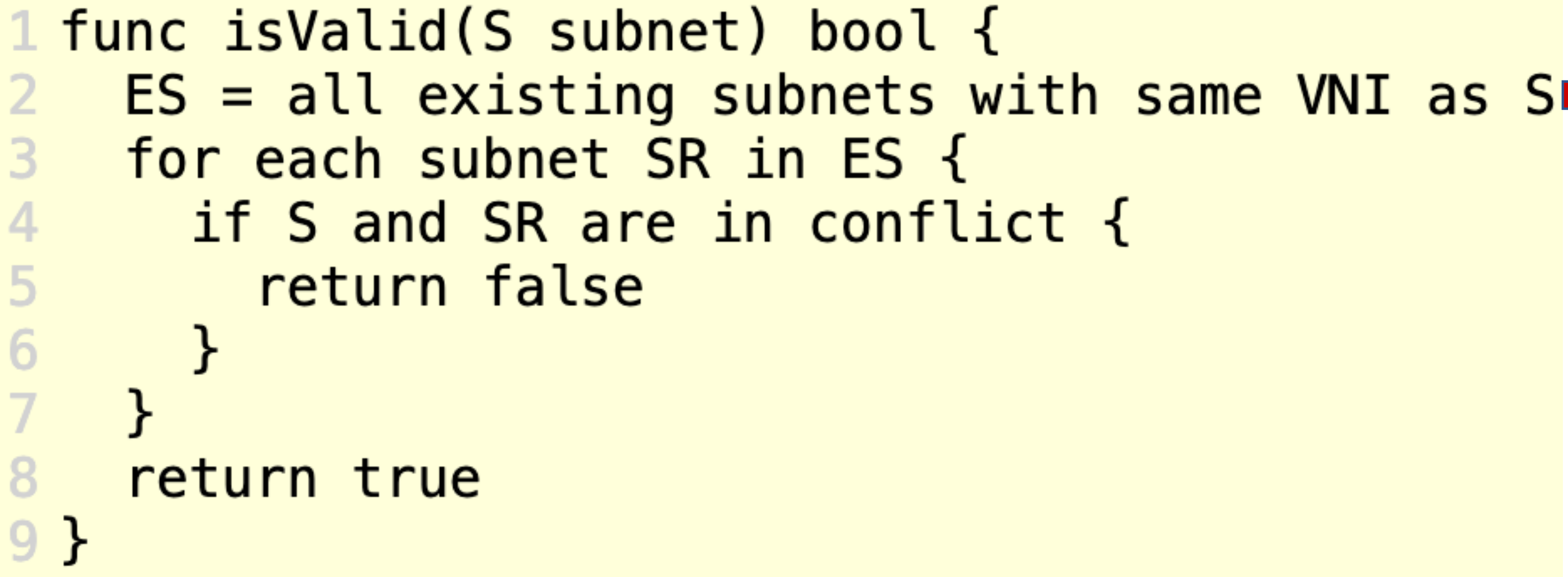
- (A): CIDR blocks MUST NOT overlap.
- (B): K8s namespaces must be the same.

Creation of a subnet that leads to a violation of (A) or (B) should fail.

So… validation must happen before creation, in API servers.

Problem of enforcing an invariant across multiple API objects of same type.

Multi-object invariants pt. 2



```
func isValid(S subnet) bool {
  ES = all existing subnets with same VNI as S
  for each subnet SR in ES {
    if S and SR are in conflict {
      return false
    }
  }
  return true
}
```

Problem: conflicting subnets could be validated in parallel.

This validation is unreliable if done before subnet is created.

Multi-object invariants pt. 3

Can’t guarantee that no conflicting subnets are created ☹.

Next best thing: if conflicting subnets exist, consumers use at most one. How?

Introduce *status.validated* field.

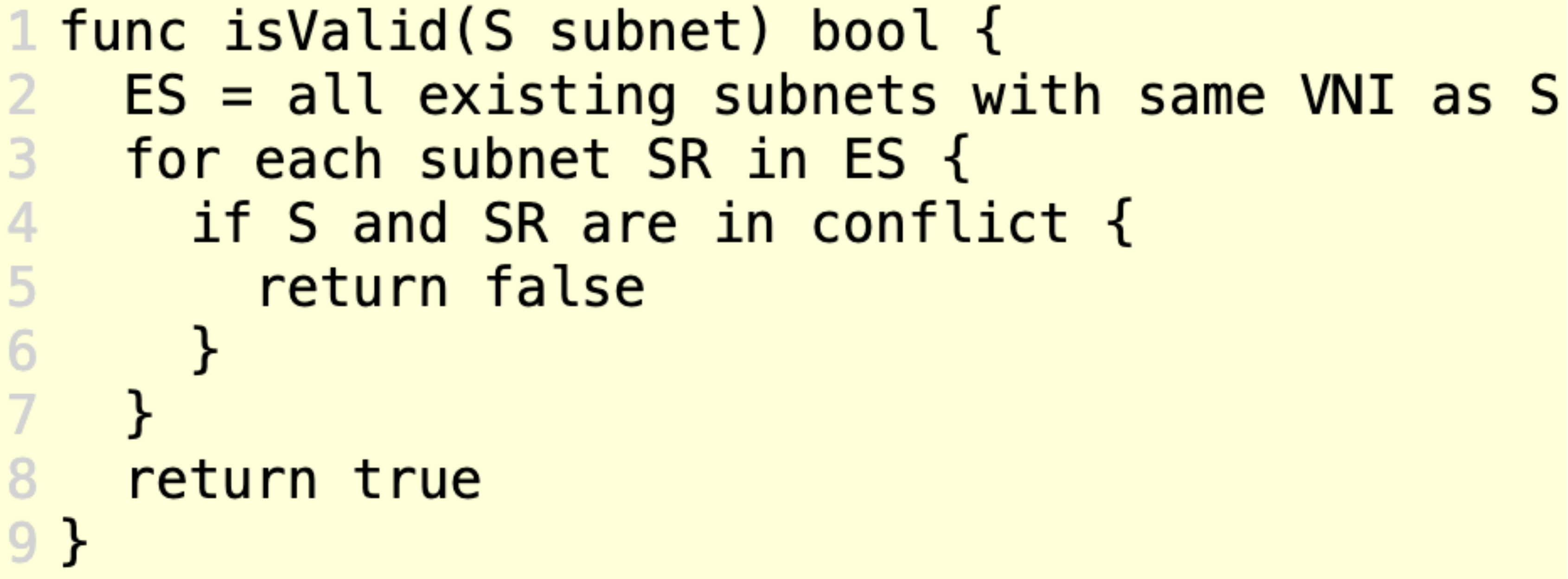
A subnet can be used only if *status.validated = true*.

If conflicting subnets exist, *at most one* has *status.validated = true*.

Multi-object invariants pt. 4

Subnet Validator:

- writes Subnets’ *status.validated*.
- singleton controller.
- has an informer on Subnets.



```
func isValid(S subnet) bool {
  ES = all existing subnets with same VNI as S
  for each subnet SR in ES {
    if S and SR are in conflict {
      return false
    }
  }
  return true
}
```

Multi-object invariants pt. 5

The Subnet Validator has one advantage over the API sever.

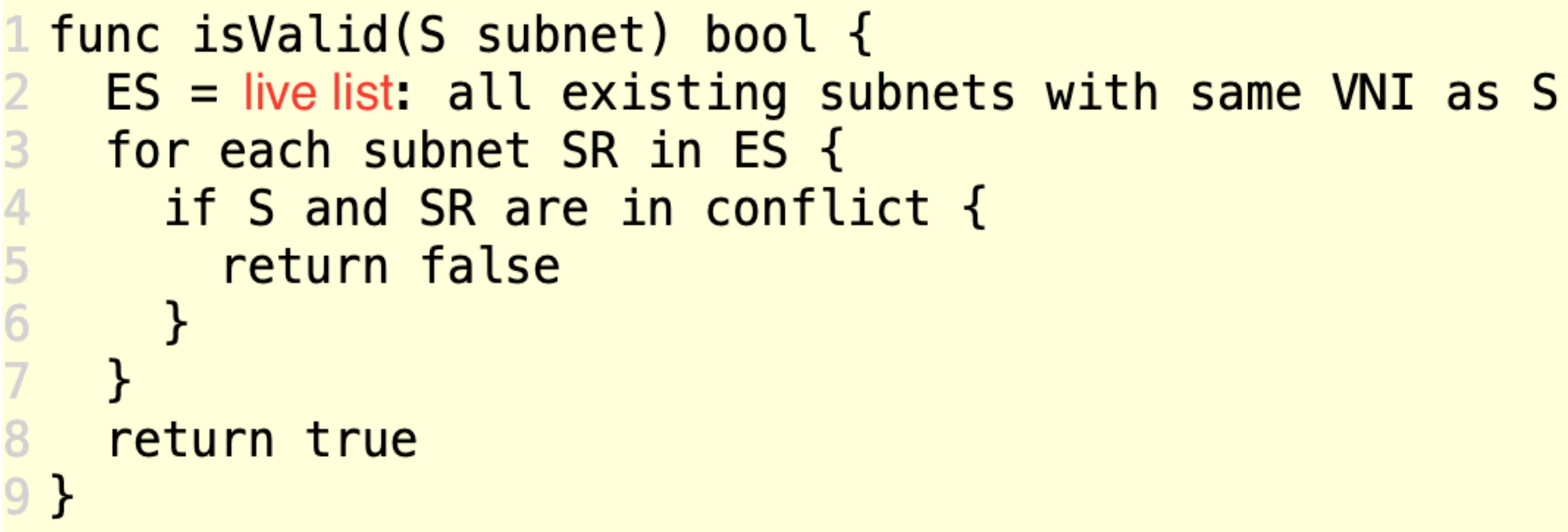
When it validates a Subnet *S1*, *S1* already exists.

API objects are created (persisted to etcd) sequentially.

If two conflicting subnets *S1* and *S2* are created, one (*S2*) is created last.

When the validator considers *S2*, *S1* already exists.

Multi-object invariants pt. 6



```
func isValid(S subnet) bool {
  ES = live list: all existing subnets with same VNI as S
  for each subnet SR in ES {
    if S and SR are in conflict {
      return false
    }
  }
  return true
}
```

Can’t retrieve existing subnets from informer cache:

- Two *Subnet Validators* might run at the same time.
- Informer caches are populated with no cross-object ordering guarantee.

Multi-object invariants pt. 7

Assume that:

- informer lists are used.
- two conflicting subnets *S1* and *S2* are created.
- two *Subnet Validators V1* and *V2* are running.

What might go wrong:

- *V1* might validate *S1 first* without seeing *S2*.
- *V2* might validate *S2 first* without seeing *S1*.

So…

Need to use live lists (against the API server): correct but less efficient.

IPAM pt. 1

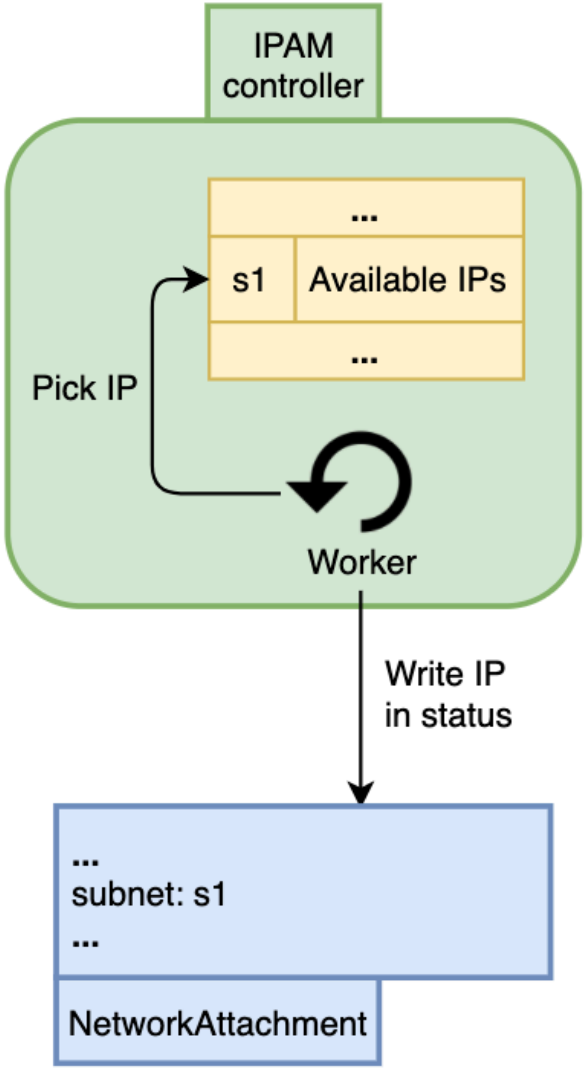
IPAM controller:

- assigns virtual IPs to *NA*s (written into *status.guestIP*).
- singleton (best-effort).

Has informers on:

- Subnets.
- *NA*s.

Picks IPs from a per-subnet, in-memory cache of available IPs.

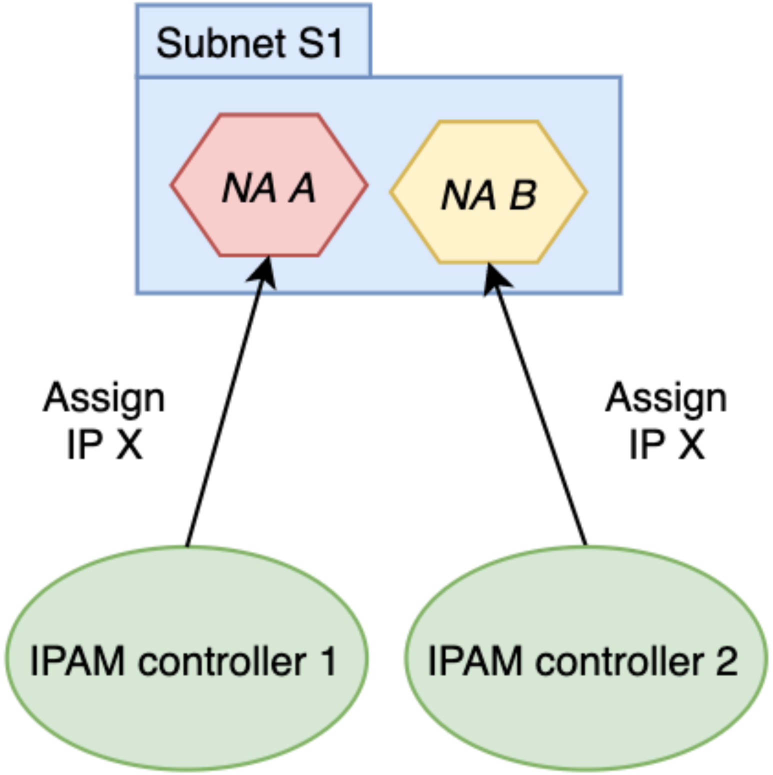


IPAM pt. 2

Multiple instances could run at once => risk of virtual IPs collisions!

So…

Local cache of available IPs not enough.



IPAM pt. 3

Solution: before assigning an IP, acquire a global lock on it.

Locking implemented as creation of an *IPLock* custom API object.

IPLock on IP *X* on VNI *Y* has name “*Y-X*”.

K8s forbids multiple API objects with same namespaced name and kind.

So…

Impossible to assign twice same IP: creation of 2nd *IPLock* fails.

Informers dynamic filtering pt. 1

Connection Agent (*CA*) on node *N* implements local *NA*s.

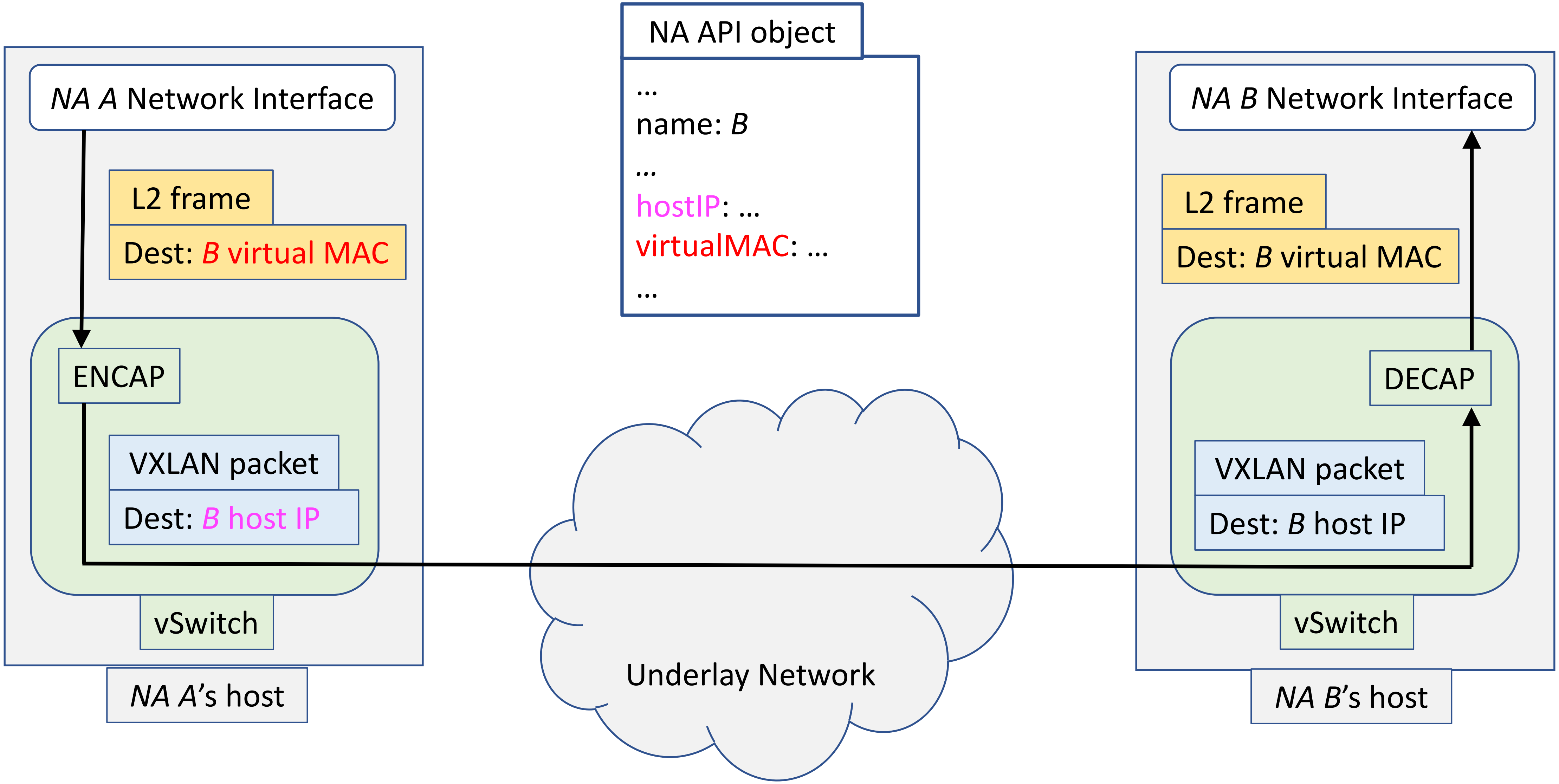
So… has informer on local *NA*s.

If *NA* with VNI *X* exists on node *N*, VNI *X* is “relevant” on *N*.

CA on *N* also needs to be notified of remote *NA*s with relevant VNIs.

Why? To ensure local *NA*s can send data to remote *NA*s.

Informers dynamic filtering pt. 2



NA API object

…

name: *B*

…

hostIP: …

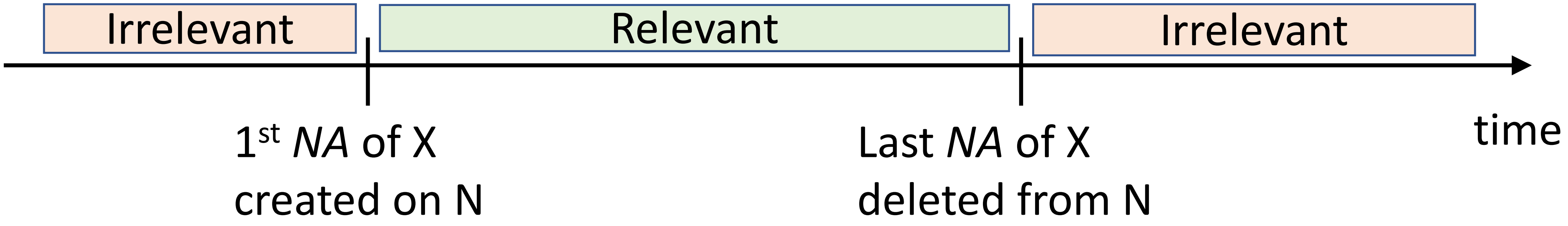
virtualMAC: …

…

Informers dynamic filtering pt. 3

Set of relevant VNIs on a node is dynamic.

Relevance of VNI *X* to Node *N:*



Informers dynamic filtering pt. 4

CA needs to be notified ONLY of remote *NA*s with relevant VNIs.

Informers support filtering.
Can use a single informer to filter on relevant VNIs? No: informer filtering is static.

How to synthesize dynamic filtering?

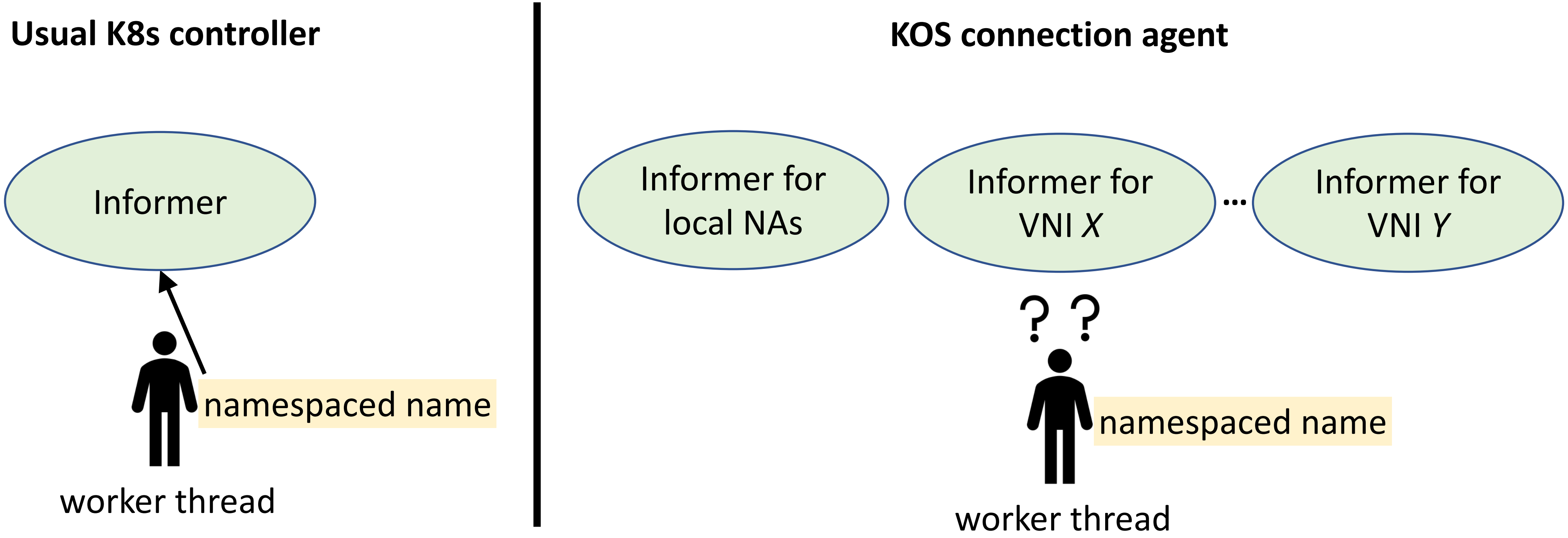
Solution: a dedicated informer for each relevant VNI.

Informers are started/stopped as VNIs become relevant/irrelevant.

Informers dynamic filtering pt. 5

A CA has potentially many informers on *NA*s.

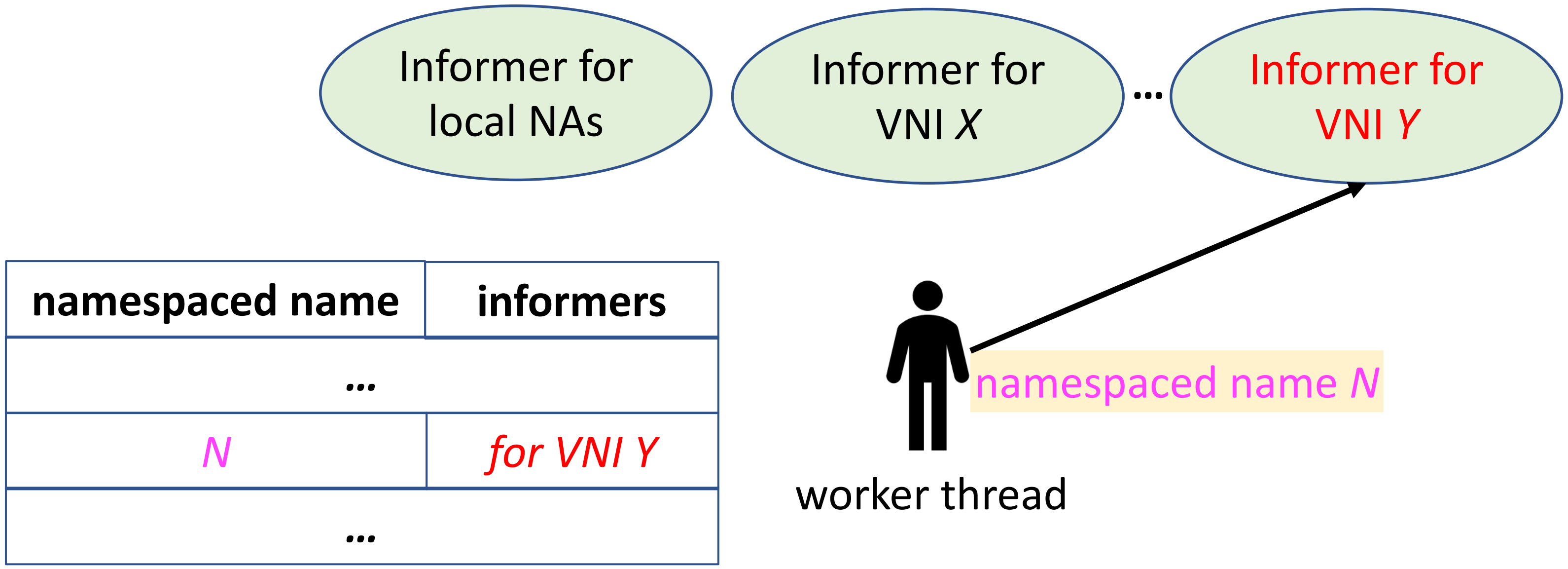
Worker threads don't know from which informer to retrieve *NA*s!



Informers dynamic filtering pt. 6

Solution: map from namespaced name to informers where the *NA* is.

The map is updated by informer notification handlers.



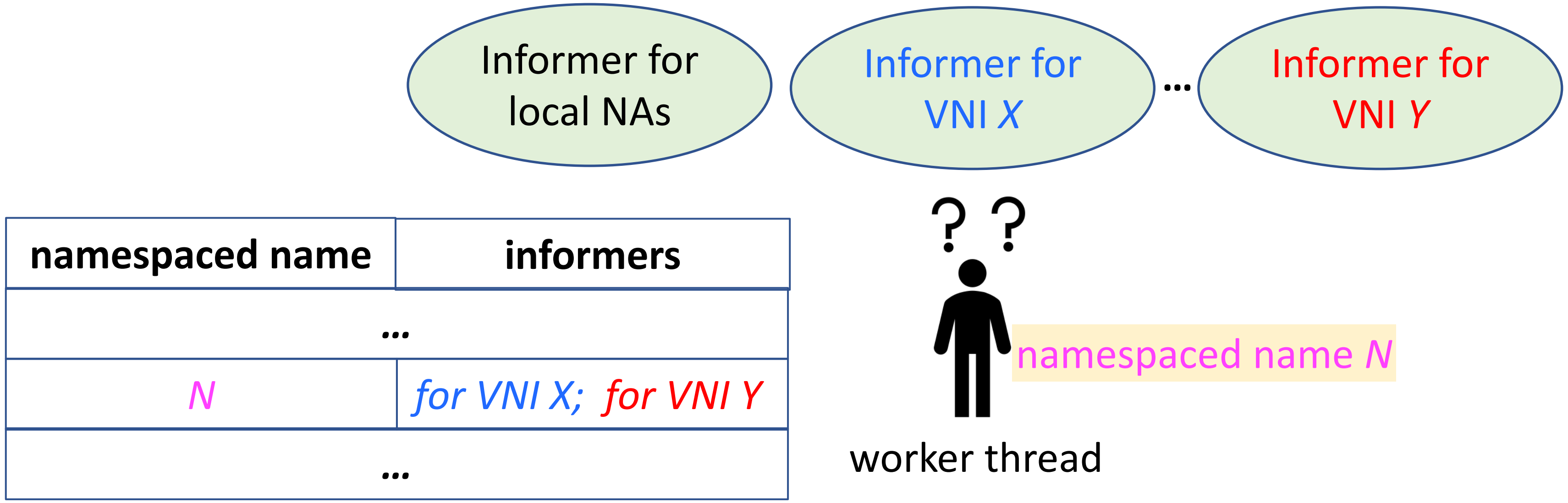
namespaced name	informers
…	
N	*for VNI Y*
…	

Informers dynamic filtering pt. 7

During transients, a *NA* can be in more than one informer, because:

- *NA* can be updated => VNI can change.
- No cross-informer ordering guarantee for notifications.

What does a worker thread do in these cases?



Informer for local NAs

Informer for VNI *X*

…

Informer for VNI *Y*

? ?

namespaced name *N*

worker thread

namespaced name	**informers**
…	
N	*for VNI X; for VNI Y*
…	

Informers dynamic filtering pt. 8

If *NA* in more than one informer, worker gives up on processing it.

Only one informer stores up-to-date version of the *NA*.

Delete notifications for old versions will come => ambiguity will be solved.

Delete notifications enqueue *NA*’s namespaced name => *NA* is re-processed.

Conclusions

We built an SDN proof of concept with K8s API machinery and controller pattern.

The result is that it can be done with relatively small effort.

Some interesting challenges emerged:

- enforcing invariants across API objects of the same type (subnets validation).
- IP assignments while avoiding IP collisions and IP leaks.
- synthesis of dynamic filtering from informers for efficient delivery of notifications.

We believe such challenges apply not just to KOS,
and required special care and creative solutions.

End

Thank you for your attention.

Q&A.

KubeCon | CloudNativeCon

Europe 2020

Virtual

KEEP CLOUD NATIVE CONNECTED