今こそ学びたい Kubernetesネットワーク

～CNIが繋ぐNWとプラットフォームの「フラッと」な対話

Takuto Nagami
@logica0419

Kotaro Kawasaki
@n4mlz

自己紹介

- Takuto Nagami (@logica0419)
- 千葉工業大学 情報科学部 情報ネットワーク学科 4年
  - 研究: **合意アルゴリズム** (Raft) × **暗号** (秘密分散)
- 得意技術: Go言語 / コンテナ / Kubernetes
- JANOG歴
  - JANOG56 **若者支援**で参加
  - JANOG57 初登壇！

自己紹介

- Kotaro Kawasaki (@n4mlz)
- 筑波大学 情報学群情報科学類 3年
- 趣味: コンテナランタイム、OS 自作、車輪の再発明
- 最近  **Cyrius** という **Linux コンテナをネイティブ実行できる自作 OS** を Rust で書いています

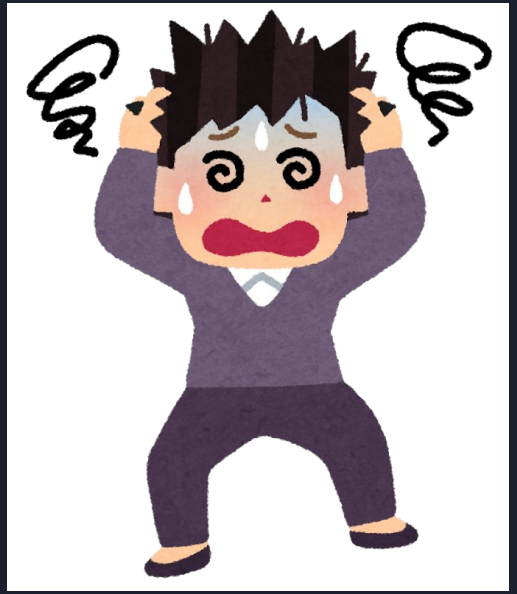
モダンなインフラ
チェックしてますか？

クラウドからクラウドネイティブへ

- ここ十数年で、**クラウド**ビジネスは大躍進を遂げた
- クラウドの設計思想は**クラウドネイティブ**へと昇華
  - クラウドで一般的になった、**スケールしやすい**開発アプローチを**あらゆる場所**で活用する
  - **オンプレ** (プライベートクラウド) でも、この思想が導入されつつある
- このクラウドネイティブなエコシステムの中心にいるのが**Kubernetes** (K8s)

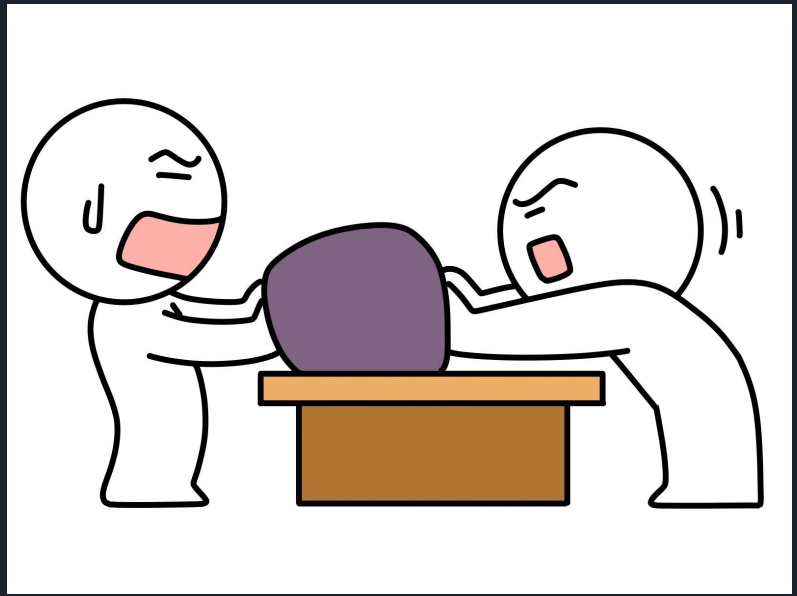
K8s、正直わからないですよね

- 「**難しい**」というイメージが先行している
- やはり「プラットフォームは**我々の管轄外**」という考えをしている人が多いのでは



プラットフォームチームも気持ちは同じ

- プラットフォーム「ネットワーク？**守備範囲じゃない**」
- ネットワーク「K8s？**守備範囲じゃない**」



ところで: 架け橋は用意されている

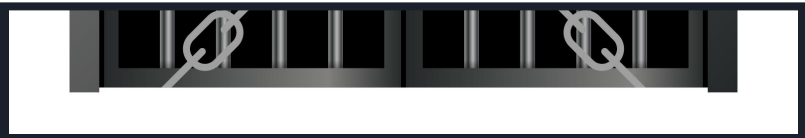
- K8sのネットワークは**差し替え可能**な設計
  - K8sのオリジナル開発者はGoogle
  - 自身のクラウドのネットワークに**最適化**できるようあえて**どんなネットワーク技術も取り込める**ように設計している
- プラットフォームとネットワークを繋ぐ**開かれた門**かつ**共通言語**の役割を担う部品がいくつかある

しかしそれらの部品の実態は…

- ネットワークもプラットフォームもお互いが忌避し、
  両者の**断絶を生む閉じた門**になっている

しかしそれらの部品の実態は…

なんて
もったいない！



OSSも銀の弾丸ではない

- **オンプレ**であっても、**OSS**を組み合わせてK8sのネットワークを構成している現場がほとんど
  - ネットワークチームは**アンダーレイのみ**管理し、K8sのネットワークは**別チーム**がOSSで運用
- OSSは**汎用性**のため、**SDN**ベース
- しかし、SDNより**物理機材**ベースの方が**効率が良い**はず
  - OSSに頼り切るのではなく、**ネットワークチームが介入**して**自作・改造**することで効率化が図れる

K8sネットワークの自社開発・改造例

- **Google Cloud**: GKE Dataplane V1 / V2
- **AWS**: Amazon VPC CNI
- **Microsoft Azure**: Azure CNI
- **Cybozu**: Coil on Neko基盤
  - 国内のクラウドでの例は**これしか見たことがない**
    (公開されていないだけかもしれません)
  - ↑ この状況を変えよう！というのが今回の目的

プラットフォームと**フラッと対話**する

- ネットワークとプラットフォームの対話なしに、真に**効率の良いネットワーク**はあり得ない
- 今回のテーマ「**フラッとJANOG**」に合わせ、フラッと気兼ねない対話に必要な**共通言語**を身につけましょう！

「プラットフォームと
ネットワークの
フラットな対話」を作る

これを今回のゴールとします！一緒に頑張りましょう

アジェンダ

- イントロダクション ←イマココ
- Kubernetesを知る
- Kubernetesとネットワーク
- Linuxのネットワークスタック
- L2/L3ネットワーク: CNI
- L4ロードバランサー: kube-proxy
- プラットフォームとの対話で広がる可能性
- まとめ

アジェンダ

- イントロダクション
- Kubernetesを知る ←イマココ
- Kubernetesとネットワーク
- Linuxのネットワークスタック
- L2/L3ネットワーク: CNI
- L4ロードバランサー: kube-proxy
- プラットフォームとの対話で広がる可能性
- まとめ

Kubernetesを**知る**

K8sは難しく思われがちだが

ポイントを絞れば

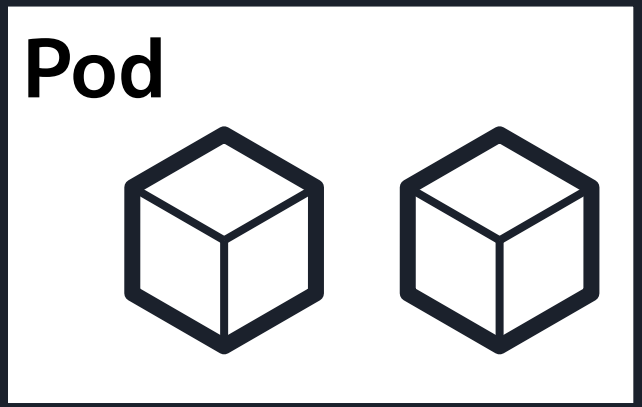
そこまで**難しくない**！

ひとまずKubernetesの世界を旅してみましょう

Kubernetesの
提供するリソース

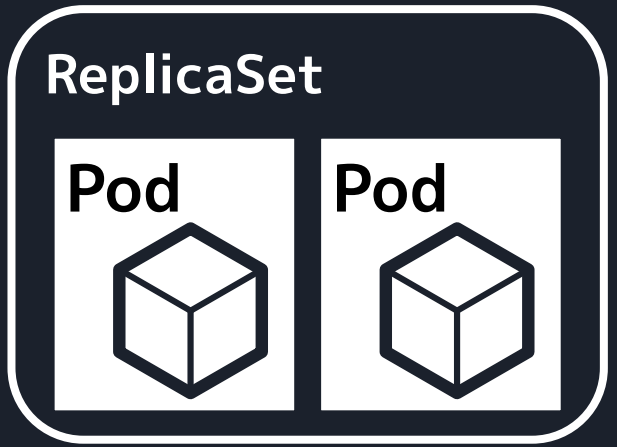
Pod

- 便利になったコンテナ、**K8sの最小単位**
- 1つのワークロードを構成する単位
  - ≒ **1つのIPアドレス**でくくって良い範囲



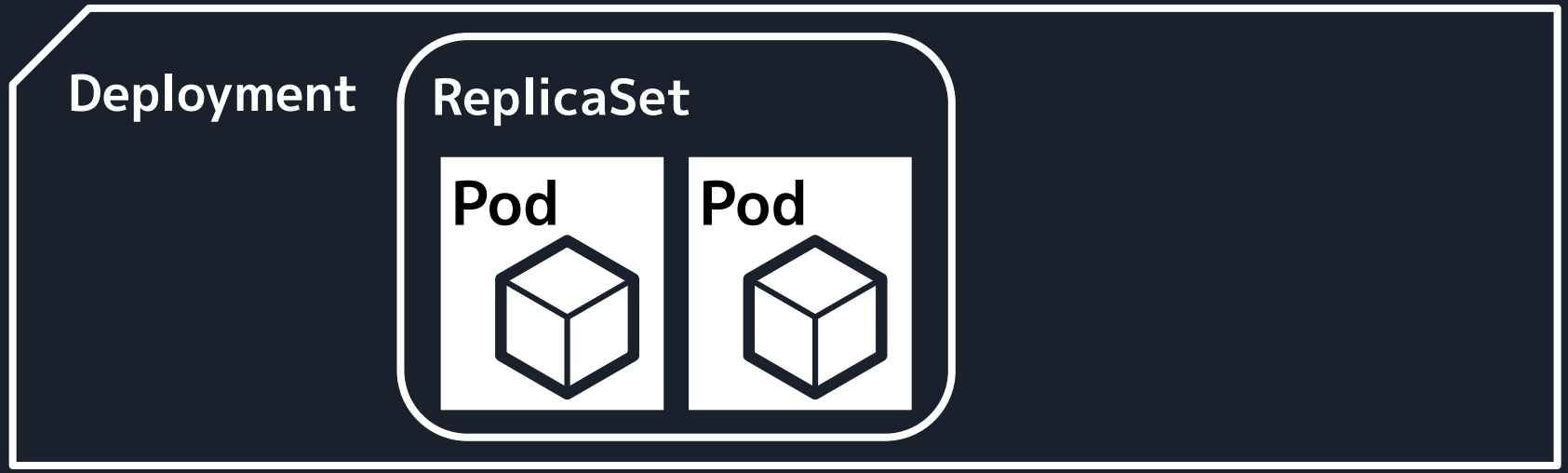
ReplicaSet / Deployment

- **Podをまとめて管理**するための仕組み
- **ReplicaSet**: 同じ内容のPodを指定個数立てる
- **Deployment**: ReplicaSetを使い安全にバージョン移行



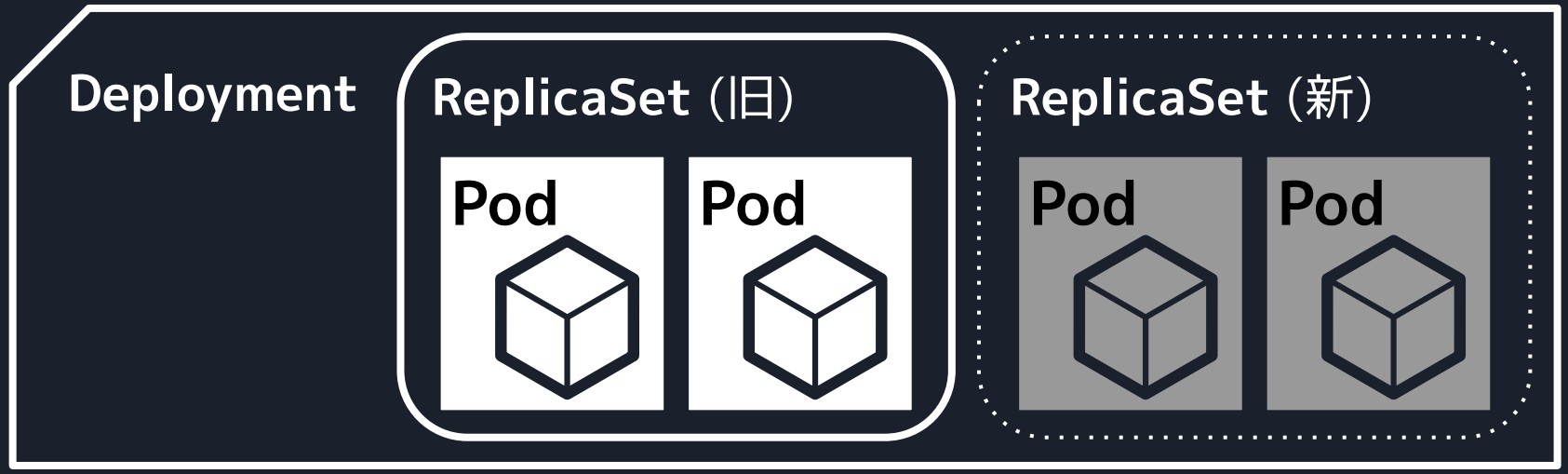
ReplicaSet / Deployment

- **Podをまとめて管理**するための仕組み
- **ReplicaSet**: 同じ内容のPodを指定個数立てる
- **Deployment**: ReplicaSetを使い安全にバージョン移行



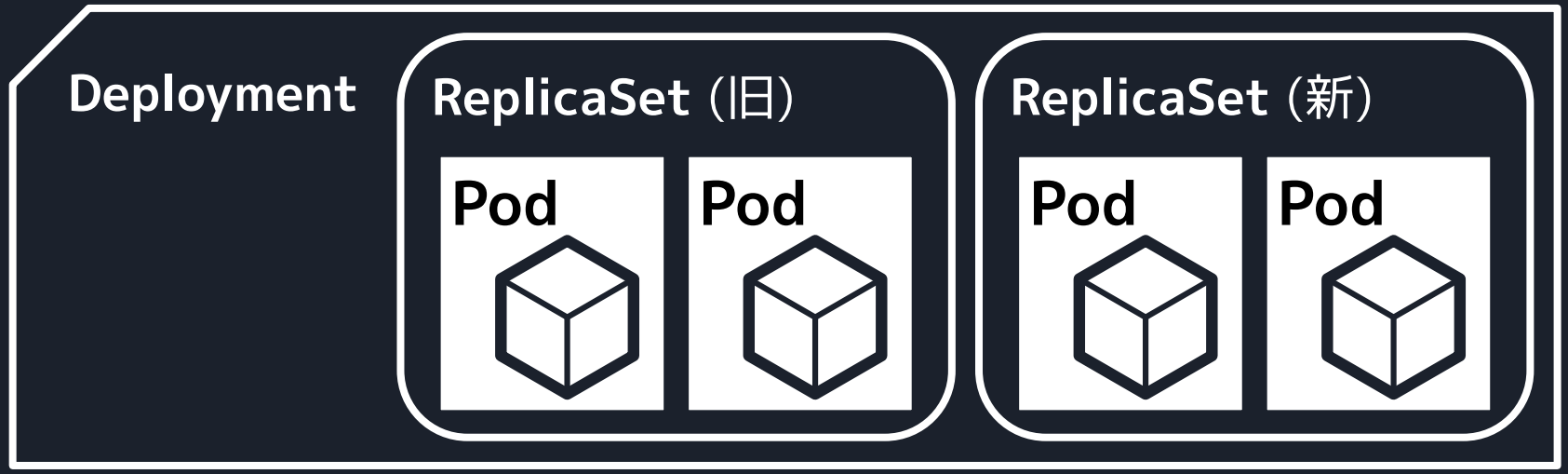
ReplicaSet / Deployment

- **Podをまとめて管理**するための仕組み
- **ReplicaSet**: 同じ内容のPodを指定個数立てる
- **Deployment**: ReplicaSetを使い安全にバージョン移行



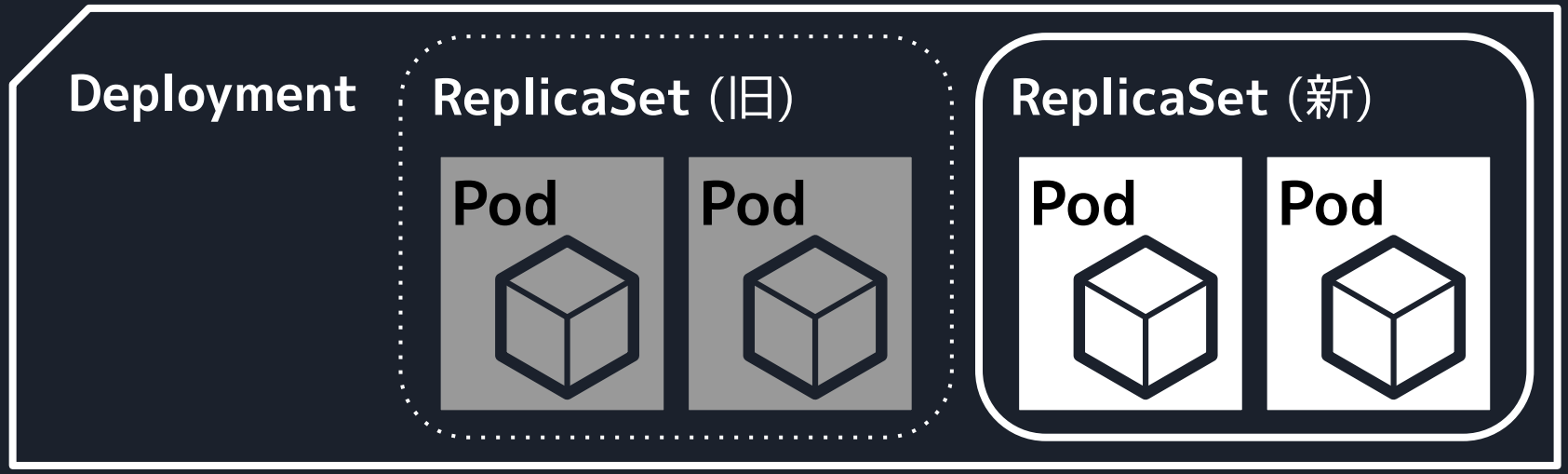
ReplicaSet / Deployment

- **Podをまとめて管理**するための仕組み
- **ReplicaSet**: 同じ内容のPodを指定個数立てる
- **Deployment**: ReplicaSetを使い安全にバージョン移行



ReplicaSet / Deployment

- **Podをまとめて管理**するための仕組み
- **ReplicaSet**: 同じ内容のPodを指定個数立てる
- **Deployment**: ReplicaSetを使い安全にバージョン移行

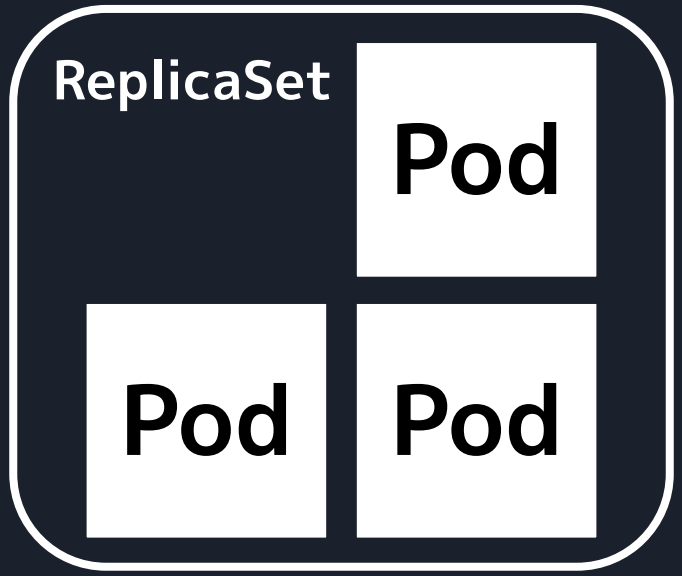


ReplicaSet / Deployment

- **Podをまとめて管理**するための仕組み
- **ReplicaSet**: 同じ内容のPodを指定個数立てる
- **Deployment**: ReplicaSetを使い安全にバージョン移行

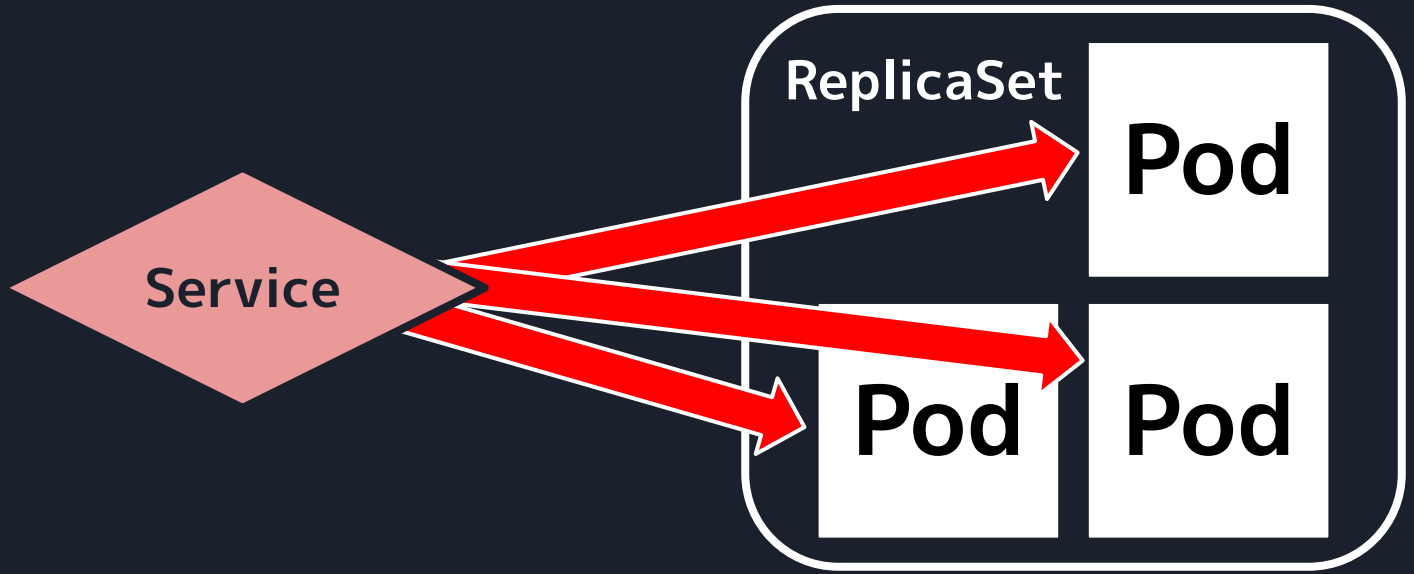
Service

- 同じ内容のPod群に、均等に**通信を振り分ける** (L4LB)
- Pod群を**1つのIPアドレスでまとめる**ことができる



Service

- 同じ内容のPod群に、均等に**通信を振り分ける** (L4LB)
- Pod群を**1つのIPアドレスでまとめる**ことができる

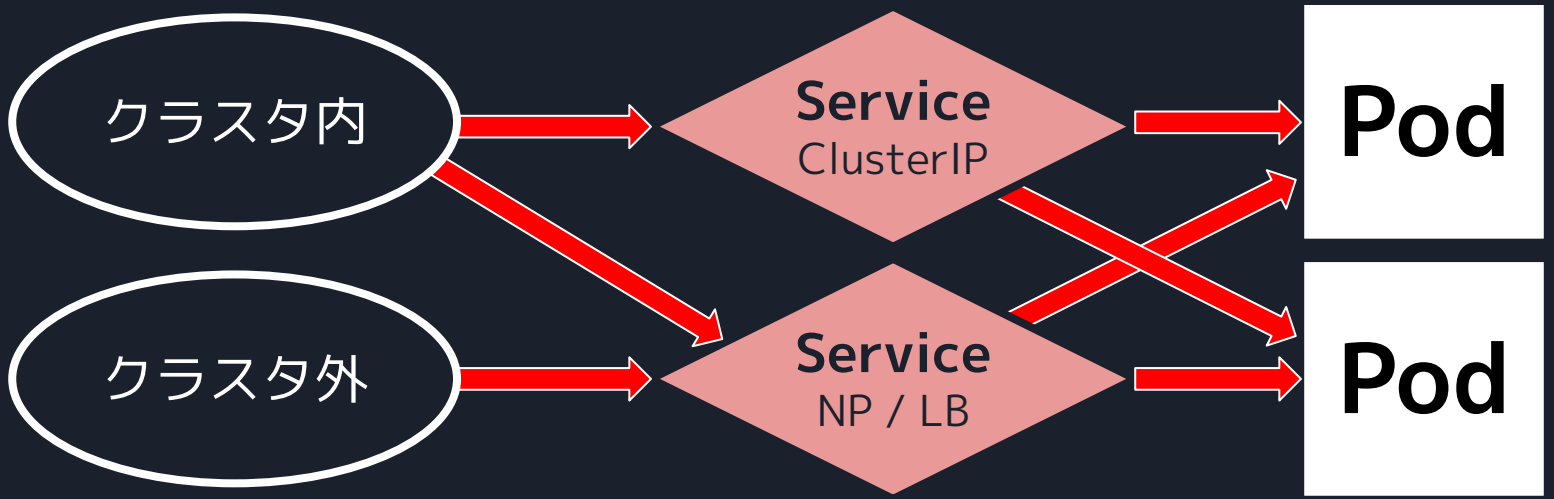


Service

Podの入れ替わりで
各Podの**IPが変わって**も
安定した仮想IPで
アクセスできる

Serviceのタイプ

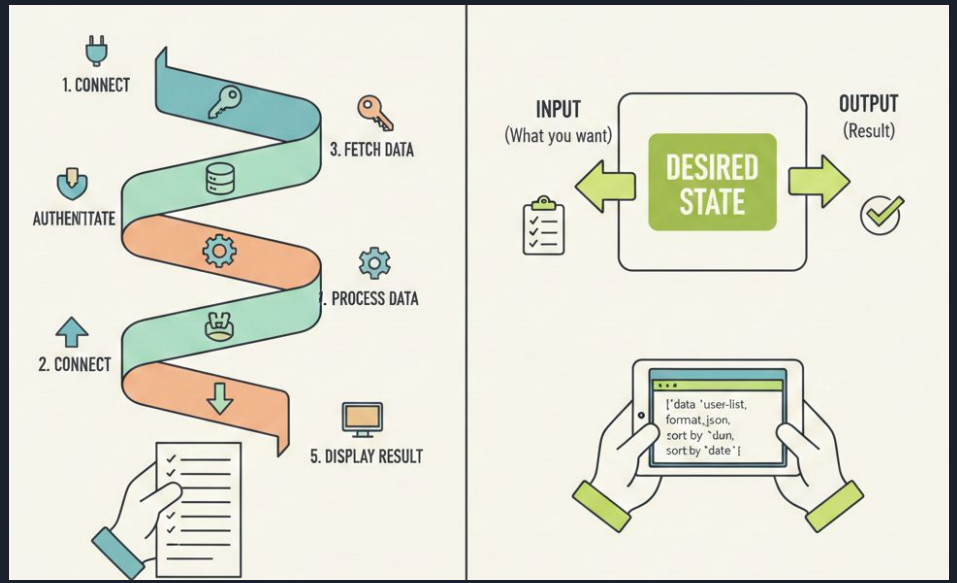
- ClusterIP: **クラスタ内**からの通信のみ受け付ける
- NodePort / LoadBalancer: **クラスタ外**からの通信も受け付ける



Kubernetesの実態:
宣言的API と
Reconciliation Loop

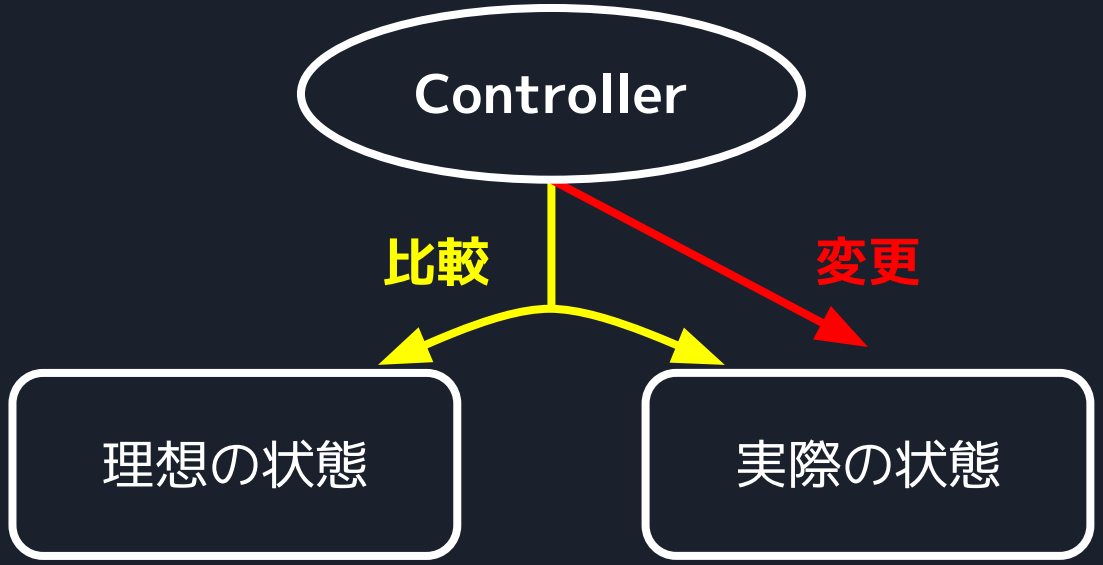
宣言的なAPI

- 「これをして下さい」でなく「**この状態にして下さい**」
  - 「マニフェスト」で**理想の状態**を書く
- 暗黙知のない
  安定した稼働



Reconciliation Loop

- リソースを**理想の状態に持っていく**ための操作

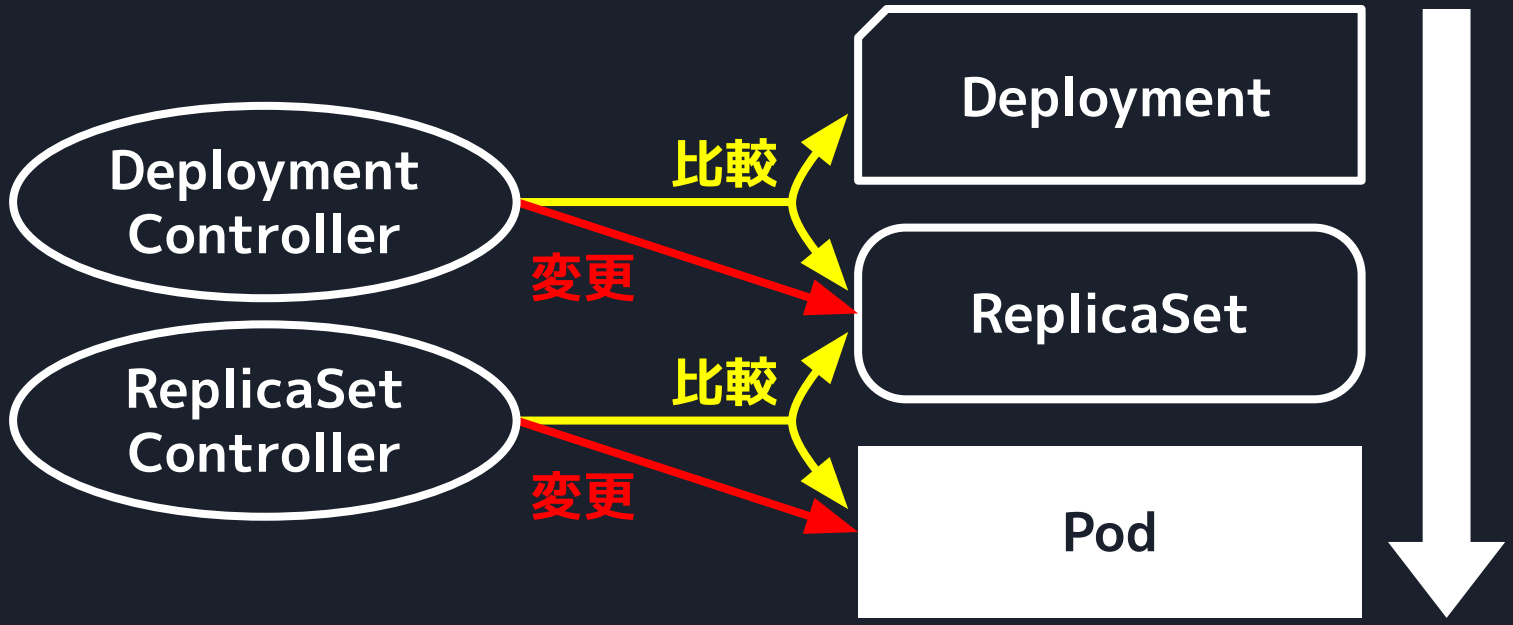


Reconciliation Loop

- 以下の操作を一定時間ごとに繰り返す
  - **理想の状態** (宣言されたマニフェスト) を取得
  - **実際の状態**を観測
  - 理想の状態と実際の状態を**比較**
  - 2つの差を埋めるように、**実際の状態を変更**する
- **Controller**と呼ばれる部品がこれを担当する
- **Self-Healing**が簡単に実装できる

例: DeploymentがPodになるまで

- 単純なReconciliation Loopの**積み重ねで複雑性**を実現



宣言された**理想の状態**を保存し**Reconciliation Loop**を回すただそれだけのシステム

ベースのアイデアは非常に単純

覚えておいてもらいたいこと

- K8sは拡張性が高いが、**ネットワークの視点**で見たとき重要なリソースは**少ない**
  - ほとんどの内部通信は、**PodからServiceを経由しPodへ**流れる
- **宣言的API**と**Reconciliation Loop**がK8sの核
  - K8sのネットワークもこの二つに準じて設定される必要がある
  - **手続き的な物を宣言的にする**実装が必要

アジェンダ

- イントロダクション
- Kubernetesを知る ←イマココ
- Kubernetesとネットワーク
- Linuxのネットワークスタック
- L2/L3ネットワーク: CNI
- L4ロードバランサー: kube-proxy
- プラットフォームとの対話で広がる可能性
- まとめ

アジェンダ

- イントロダクション
- Kubernetesを知る
- Kubernetesとネットワーク ←イマココ
- Linuxのネットワークスタック
- L2/L3ネットワーク: CNI
- L4ロードバランサー: kube-proxy
- プラットフォームとの対話で広がる可能性
- まとめ

Kubernetesと
ネットワーク

Kubernetesを流れる通信 3種

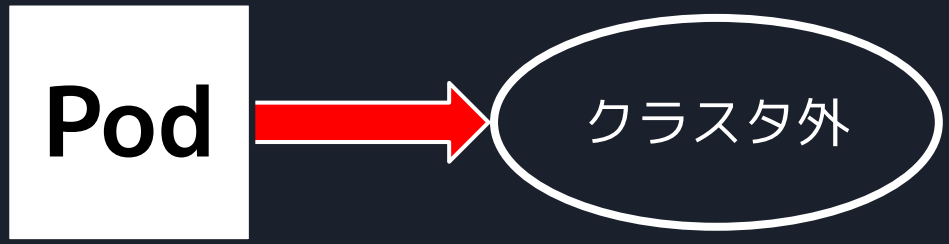
- **Pod → Service → Pod**
  - **クラスタ内**通信
  - **ノード内**と**ノード間**を考慮する必要がある

Kubernetesを流れる通信 3種

- **外部 → Service → Pod**
  - **クラスタ外から**のIngress通信
  - **Gateway API** (L7LB) というリソースとの関わりが深いが、本プログラムでは割愛

Kubernetesを流れる通信 3種

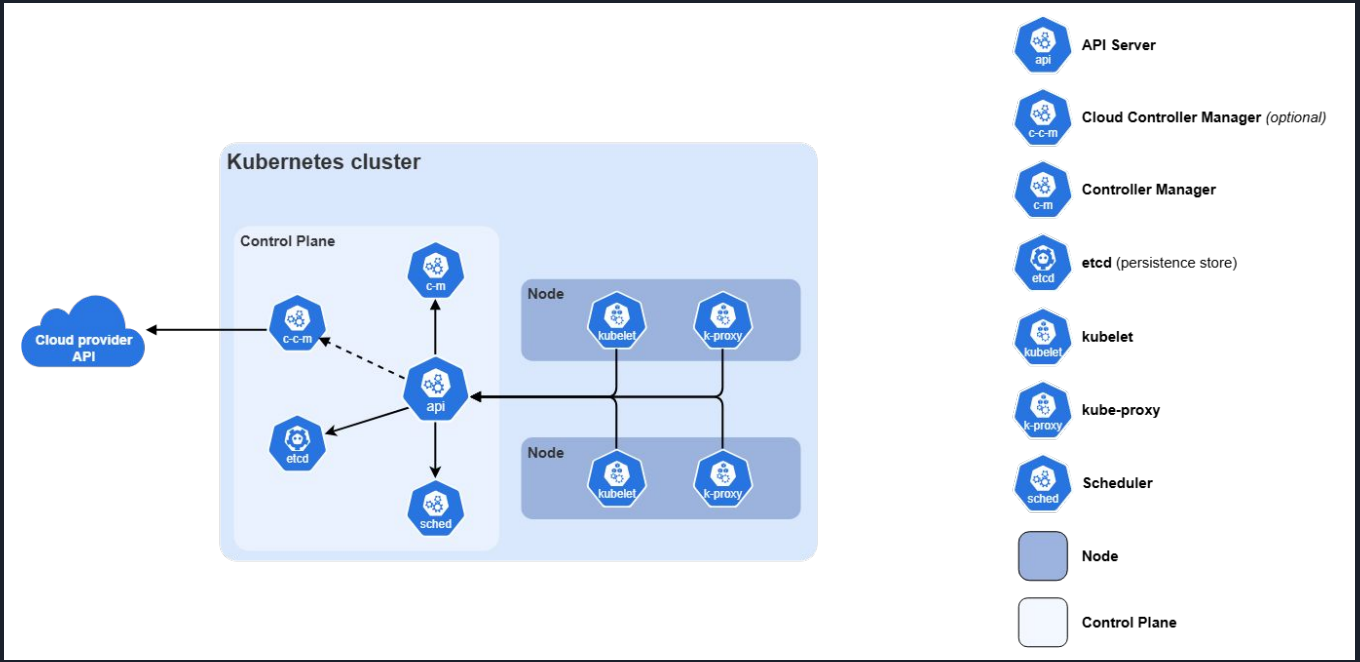
- **Pod → 外部**
  - **クラスタ外**へのEgress通信
  - Podからの**インターネット疎通**を保証する



k

Kubernetesを構成する部品たち

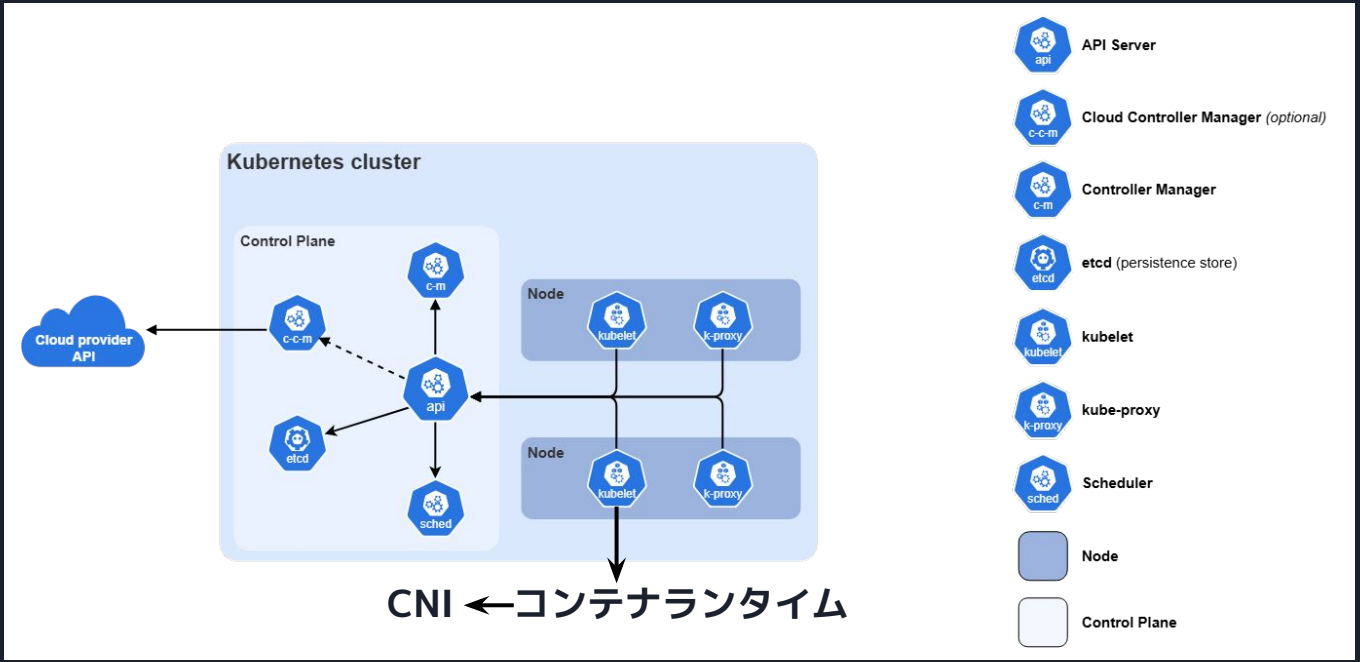
https://kubernetes.io/docs/concepts/overview/components/ より引用



k

Kubernetesを構成する部品たち

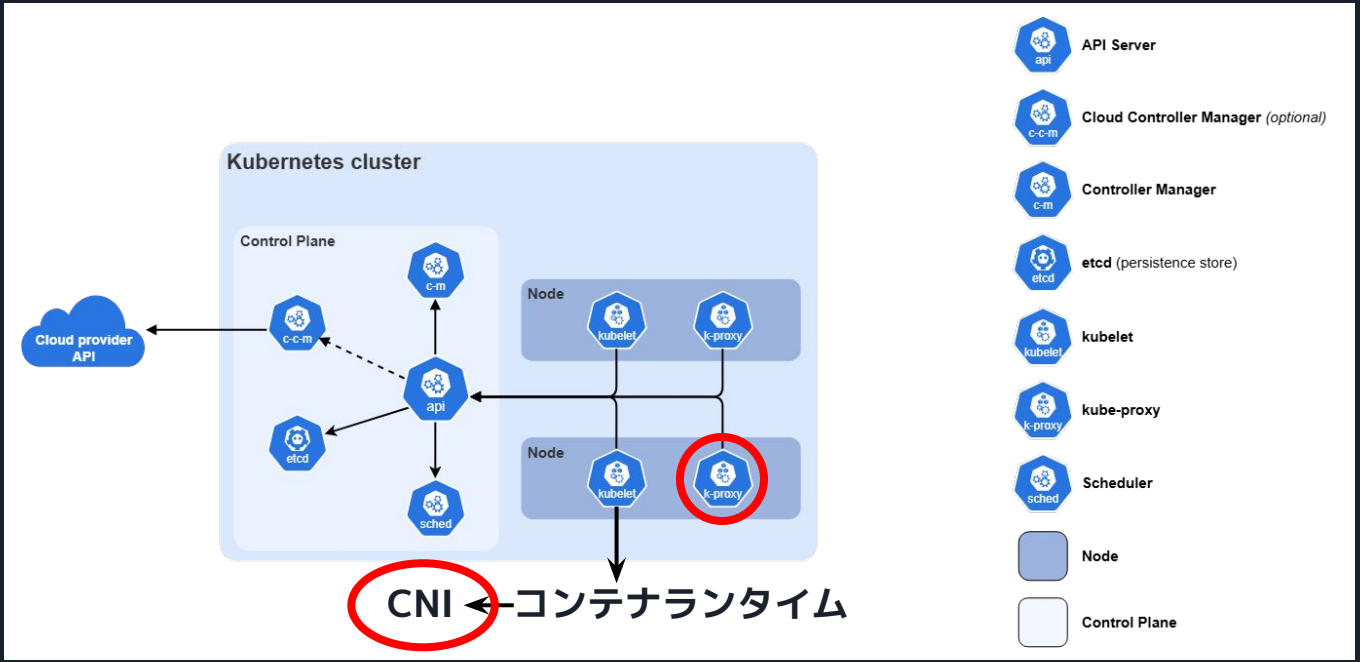
- K8sが**デフォルトで用意してない**ものもある



k

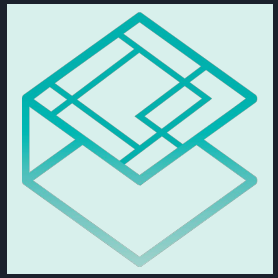
ネットワークを担当する部品たち

- **CNI と kube-proxy**



ネットワークを担当する部品たち

- **CNI**
  - **Pod → Pod / Pod → 外部** を担当
  - Podの**L2/L3ネットワーク疎通**を担保する
  - Kubernetesのデフォルト実装は**ない**
- **kube-proxy**
  - **Service → Pod / 外部 → Service** を担当
  - 複数Podに対する**L4LB** (Service) を実装する
  - Kubernetesがデフォルト実装を**用意している**



ここからは
それぞれの部品の**詳細**と
実装を見ていきます

どのように「架け橋」が用意されているかにご注目下さい

アジェンダ

- イントロダクション
- Kubernetesを知る
- Kubernetesとネットワーク ←イマココ
- Linuxのネットワークスタック
- L2/L3ネットワーク: CNI
- L4ロードバランサー: kube-proxy
- プラットフォームとの対話で広がる可能性
- まとめ

アジェンダ

- イントロダクション
- Kubernetesを知る
- Kubernetesとネットワーク
- Linuxのネットワークスタック **←イマココ**
- L2/L3ネットワーク: CNI
- L4ロードバランサー: kube-proxy
- プラットフォームとの対話で広がる可能性
- まとめ

Linuxの
ネットワークスタック

それぞれの詳細を見る前に
まずは**前提知識**となる
Linuxのネットワークをおさらい

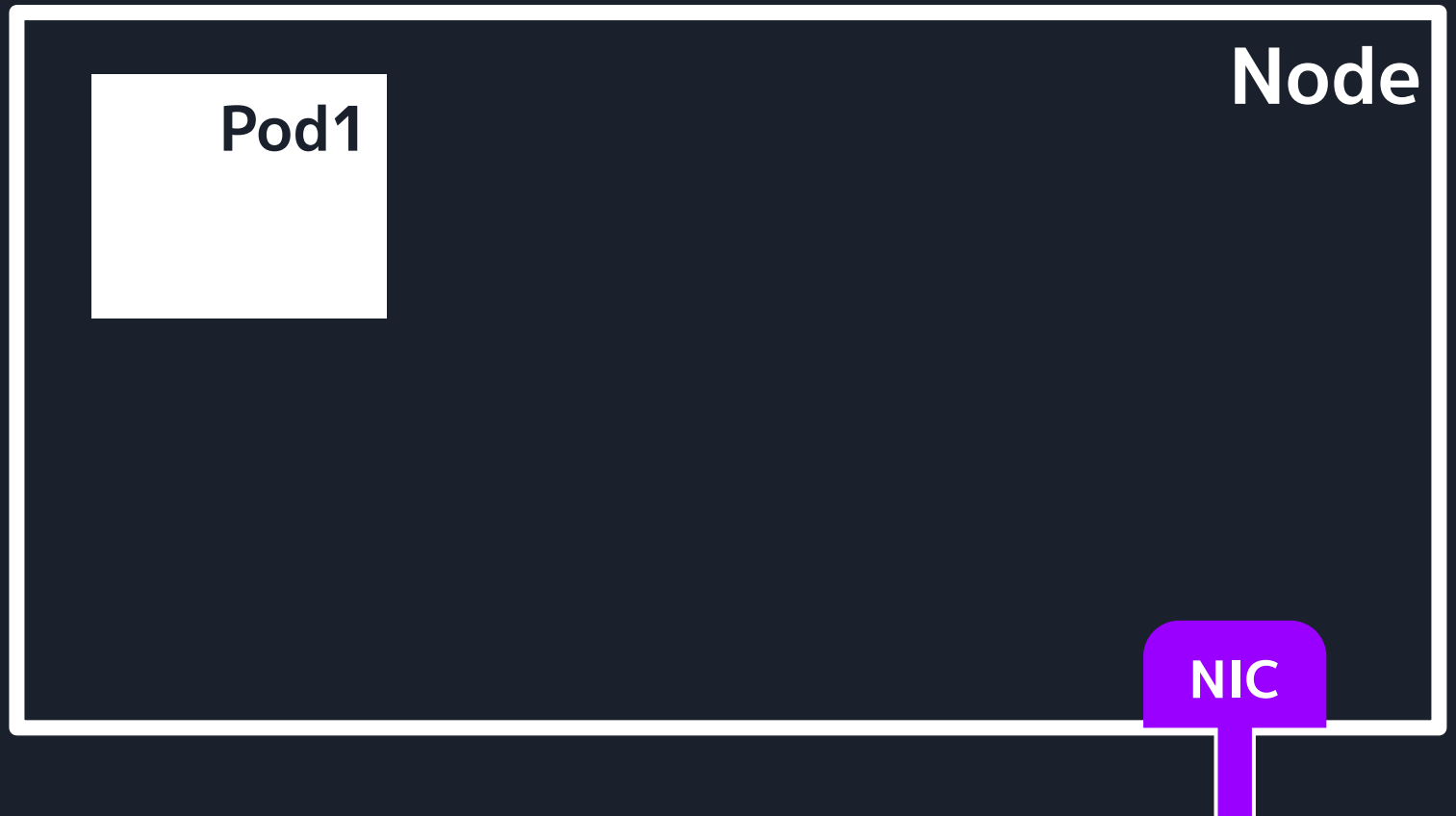
LinuxのSDNを一度俯瞰してみましょう

Network Namespace (netns)

- Linuxの機能のうちの一つ
- 1台のマシンの中に**独立した仮想のホスト**を作る
- **netns1つにつき、1マシンと**考えてよい
  - **Podの** (ネットワーク的な) **実体**
  - それぞれ個別に、NICを生やしたりIP/MACアドレスを持たせたりできる

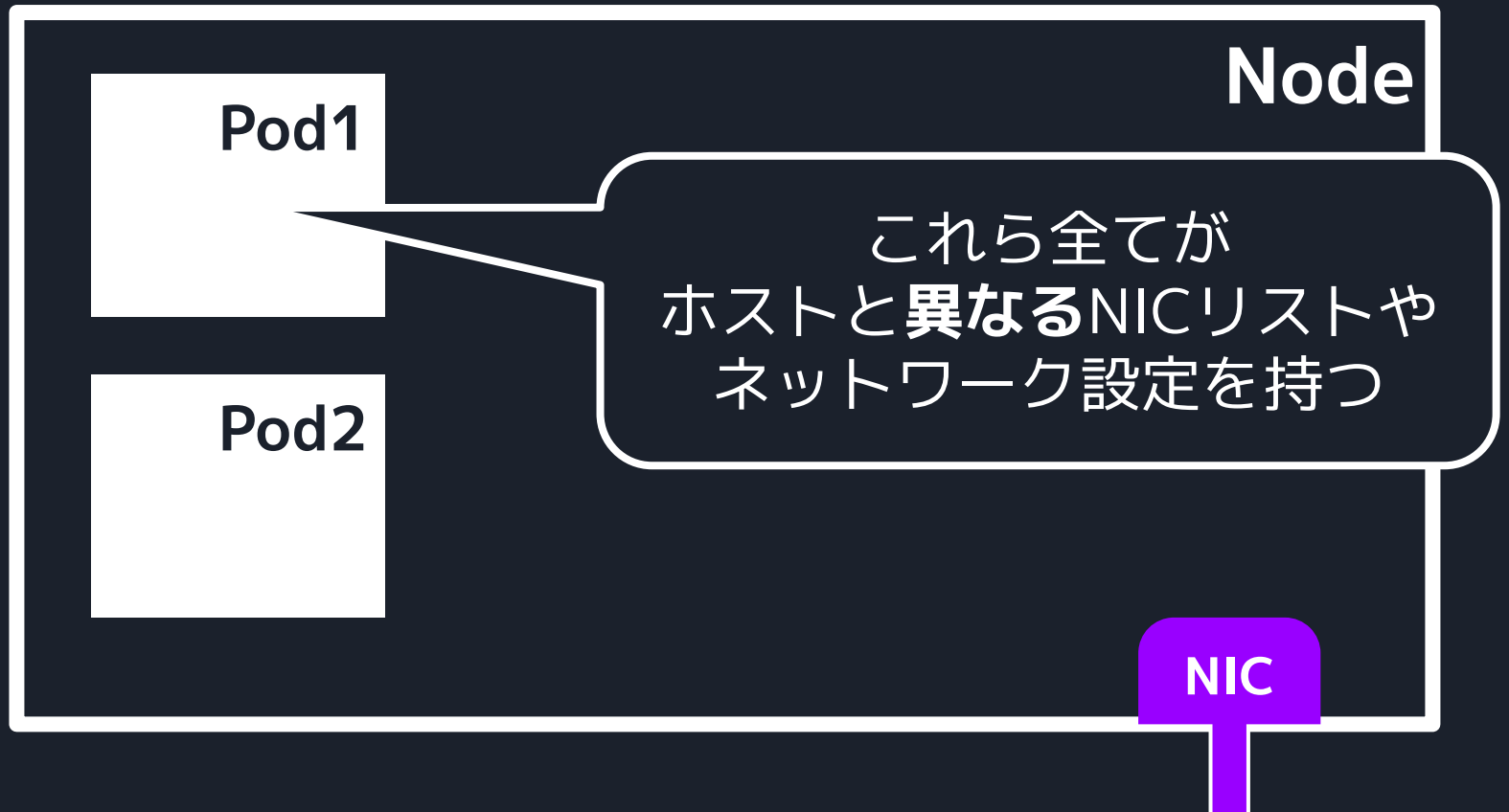
Network Namespace (netns)

Network Namespace (netns)



Network Namespace (netns)

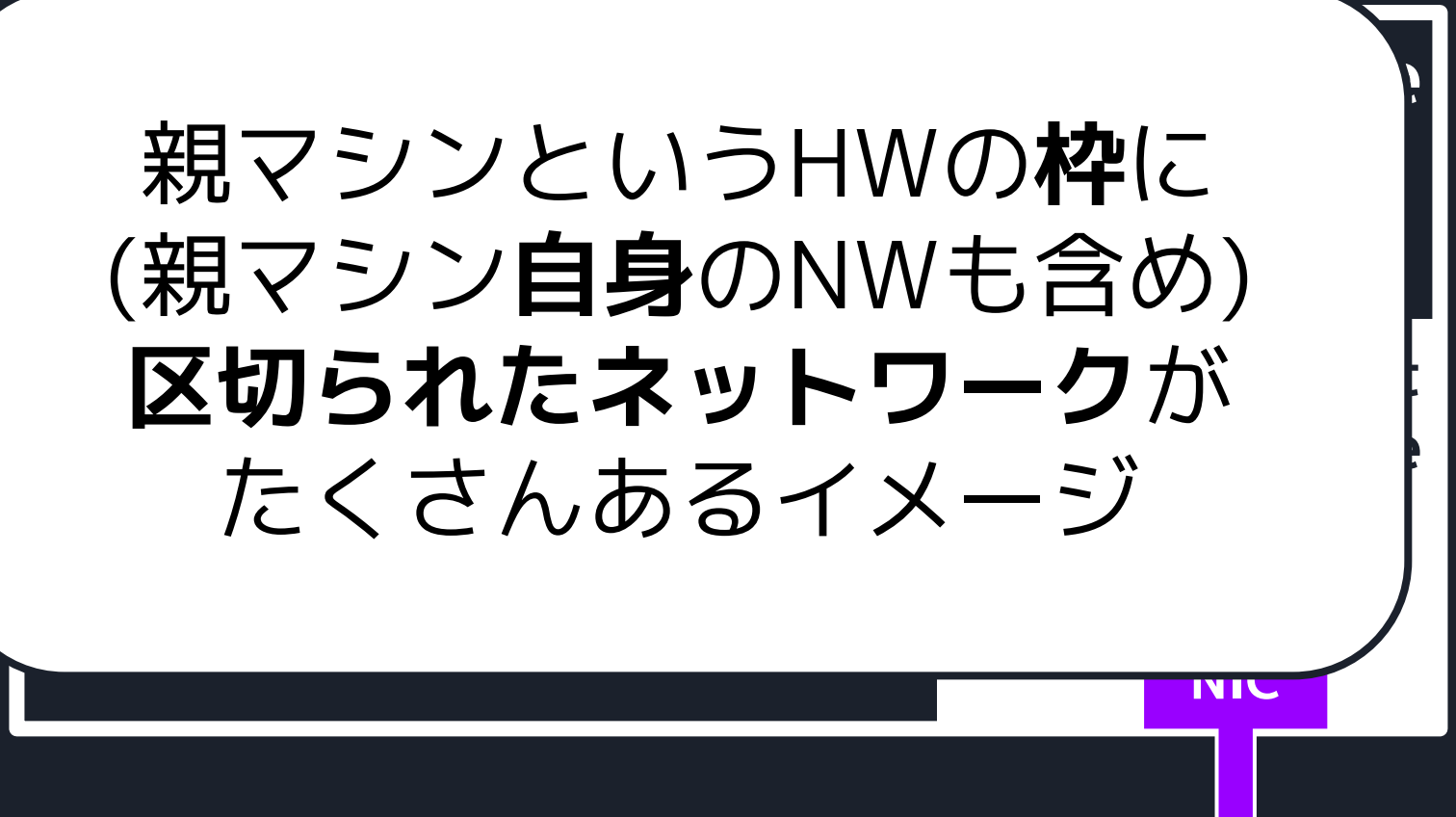
Network Namespace (netns)



実は、親マシンのネットワークも…

実は、親マシンのネットワークも…

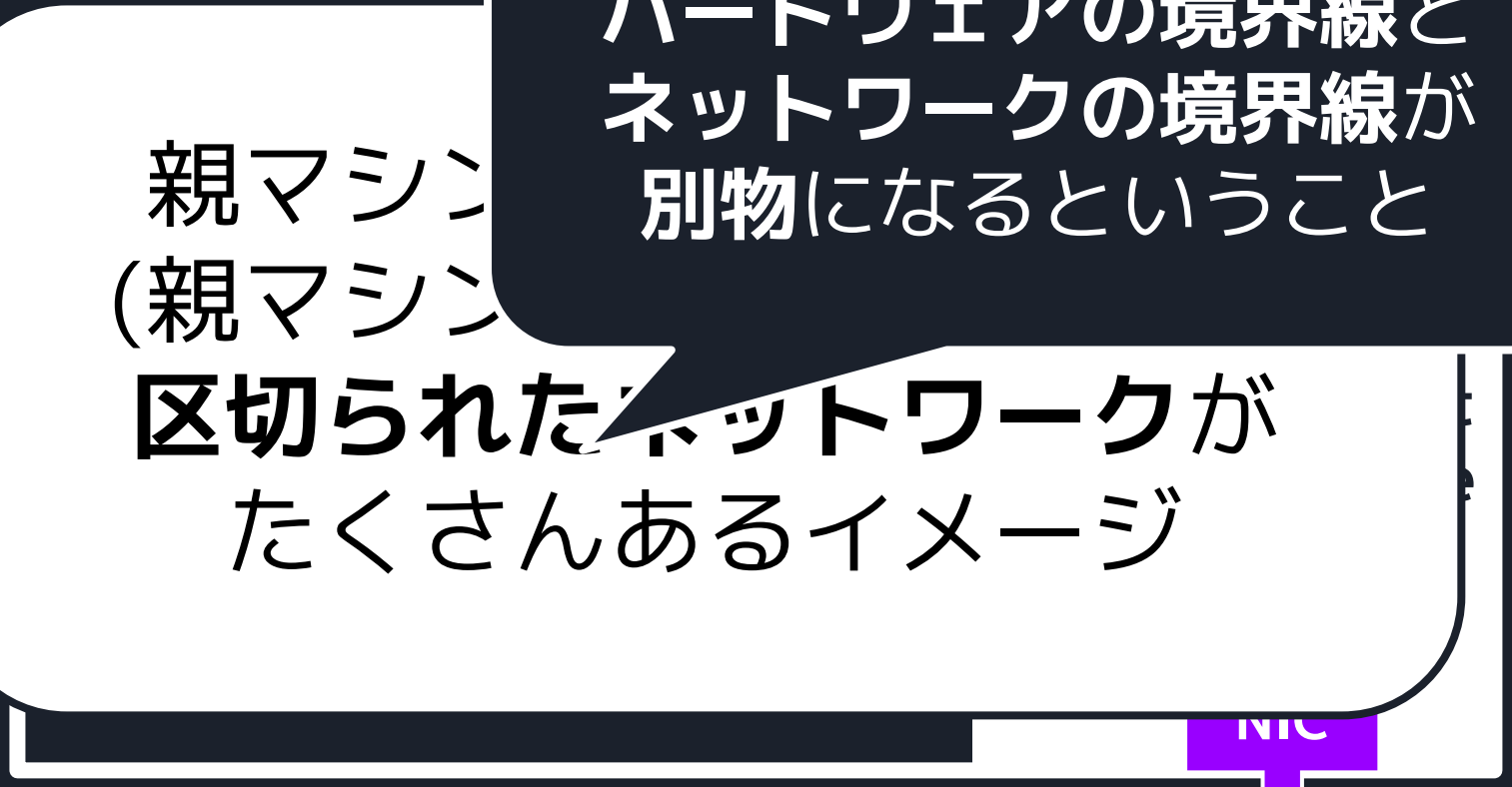
親マシンというHWの**枠**に
(親マシン**自身**のNWも含め)
区切られたネットワークが
たくさんあるイメージ



実は、親マシ

親マシン
(親マシン
区切られたネットワークが
たくさんあるイメージ

ハードウェアの境界線と
ネットワークの境界線が
別物になるということ

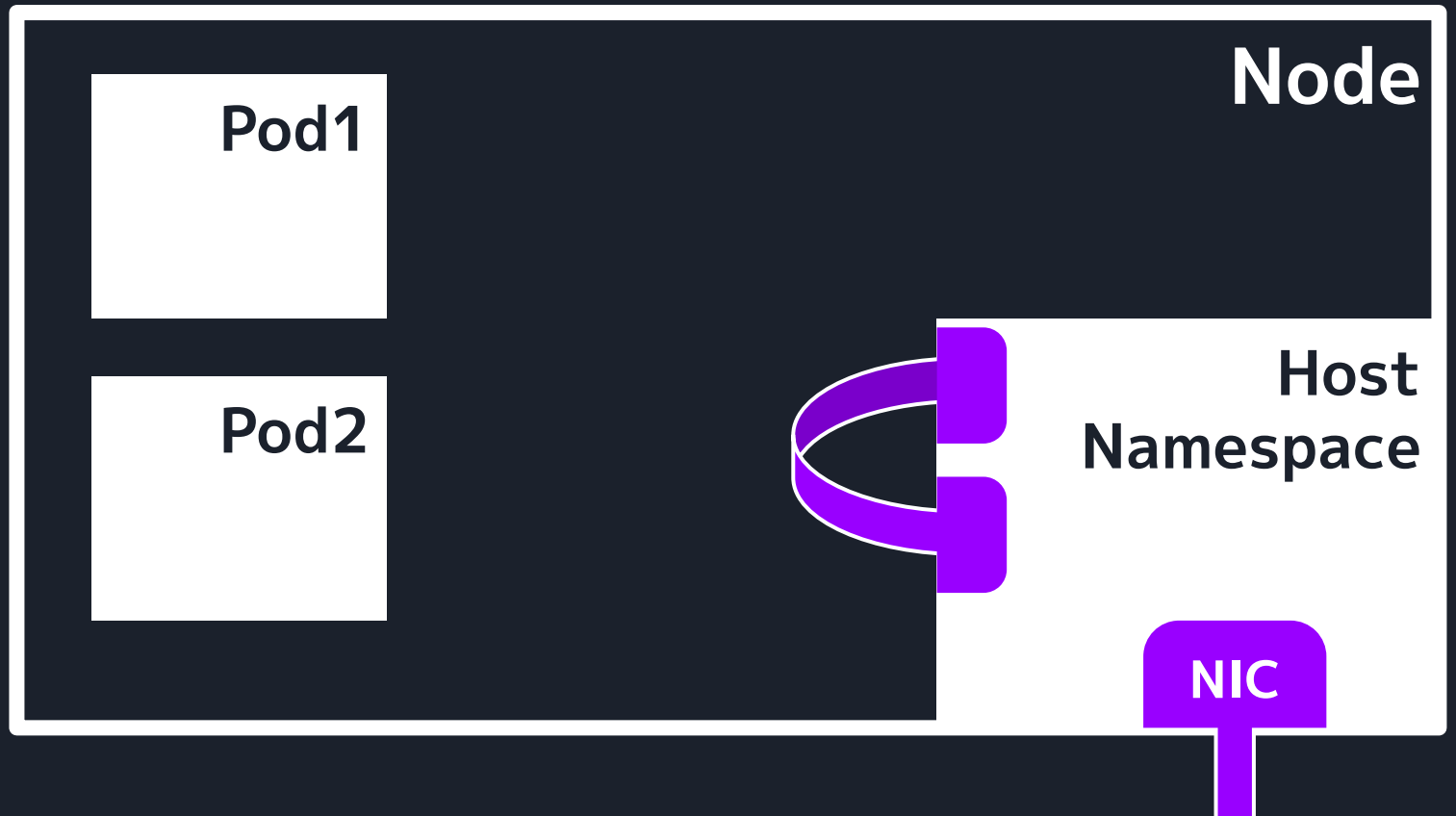


veth

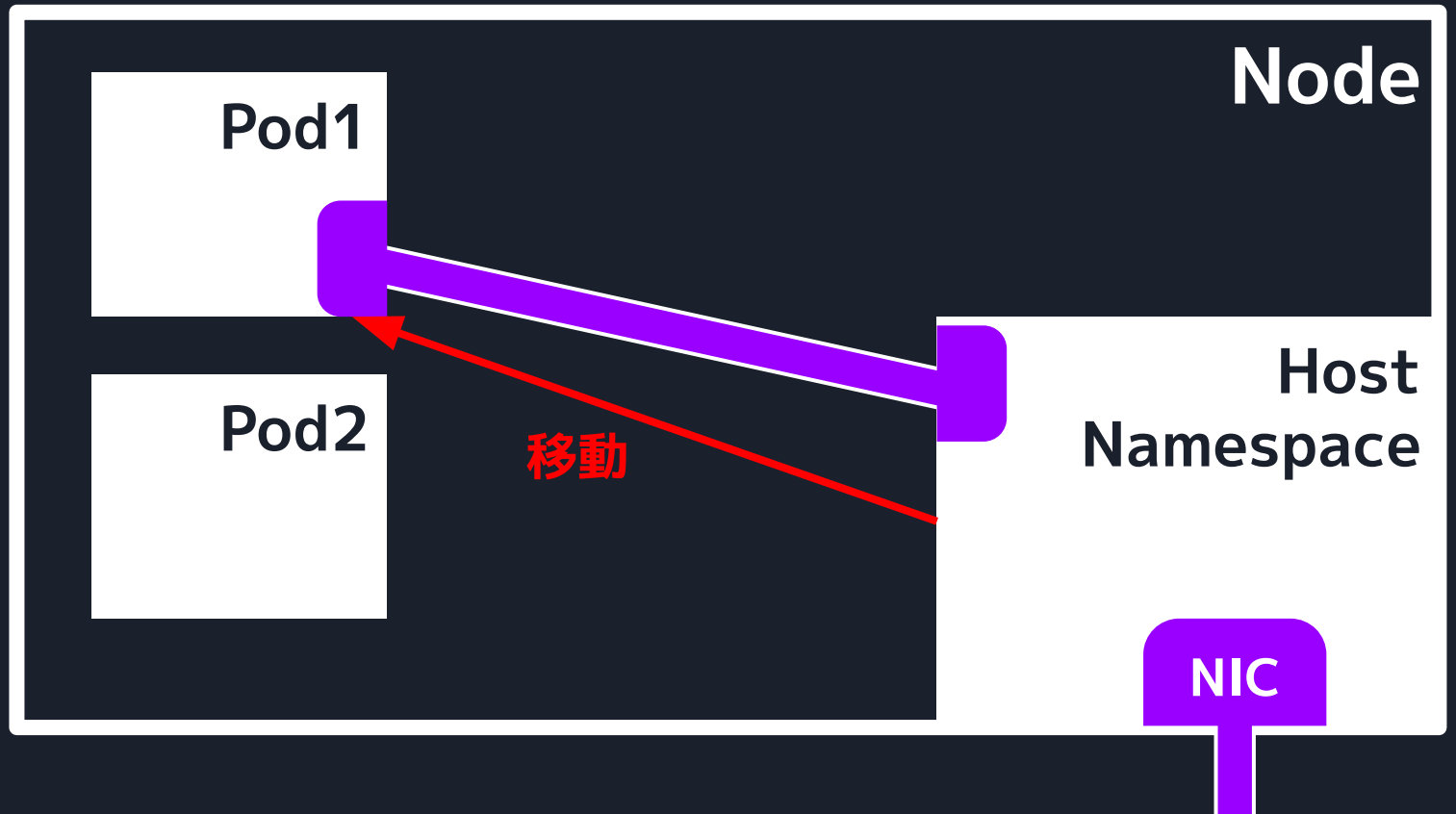
- Linux内で**仮想的**に作られるイーサネットケーブル
  - 繋がった**NICのペア**として生成される
- これによりnetns同士を接続することが可能になる

veth

veth

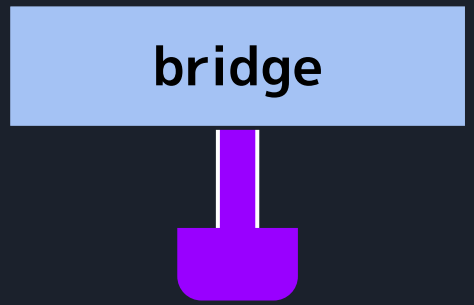


veth



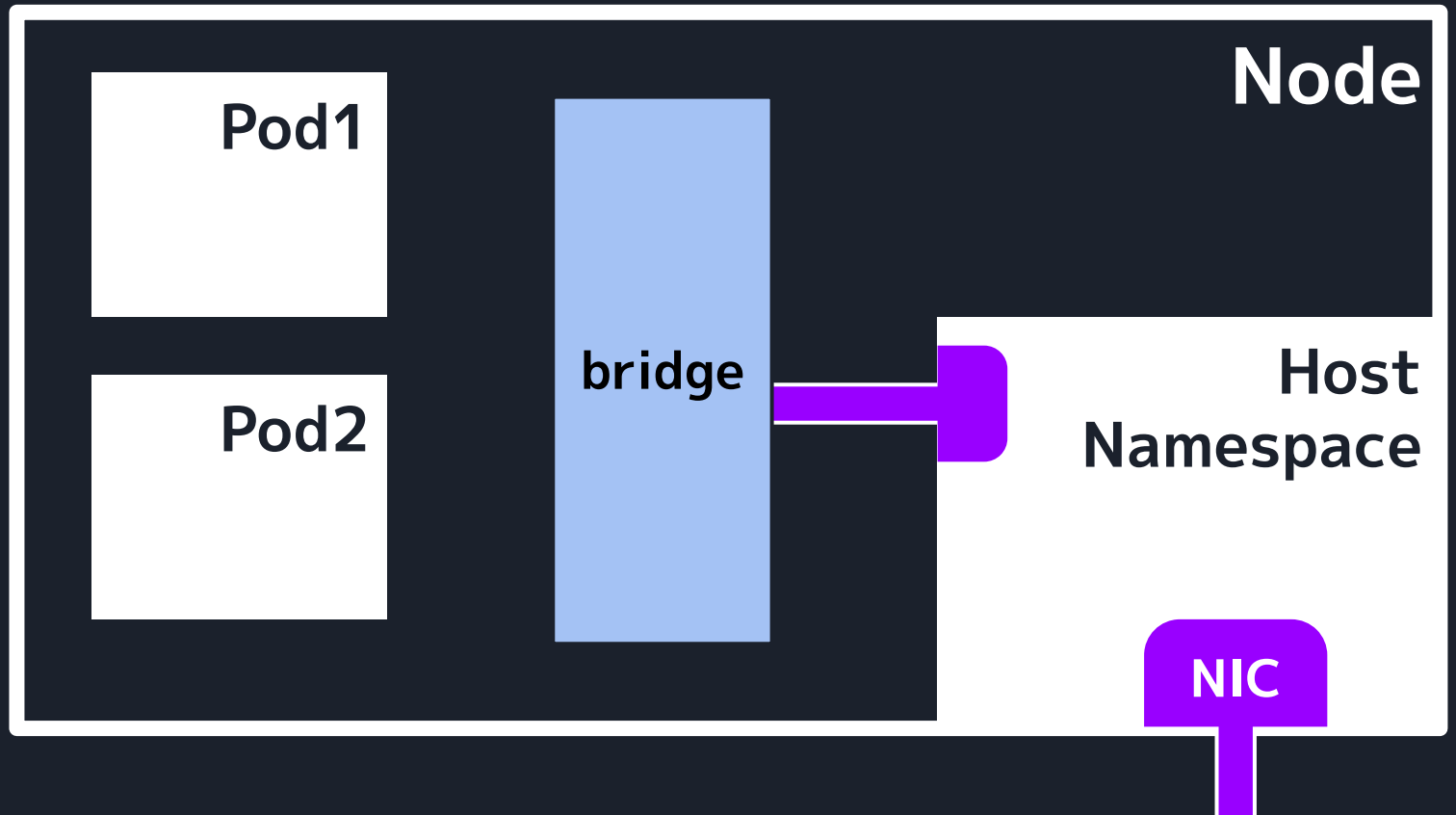
Linux Bridge

- 仮想**L2スイッチ**
  - 複数のvethを接続すると、接続された**netns同士の Ethernet疎通・ARP解決**ができるようになる
- Linuxの都合で、**host namespaceと接続されたNIC**があらかじめ生成される

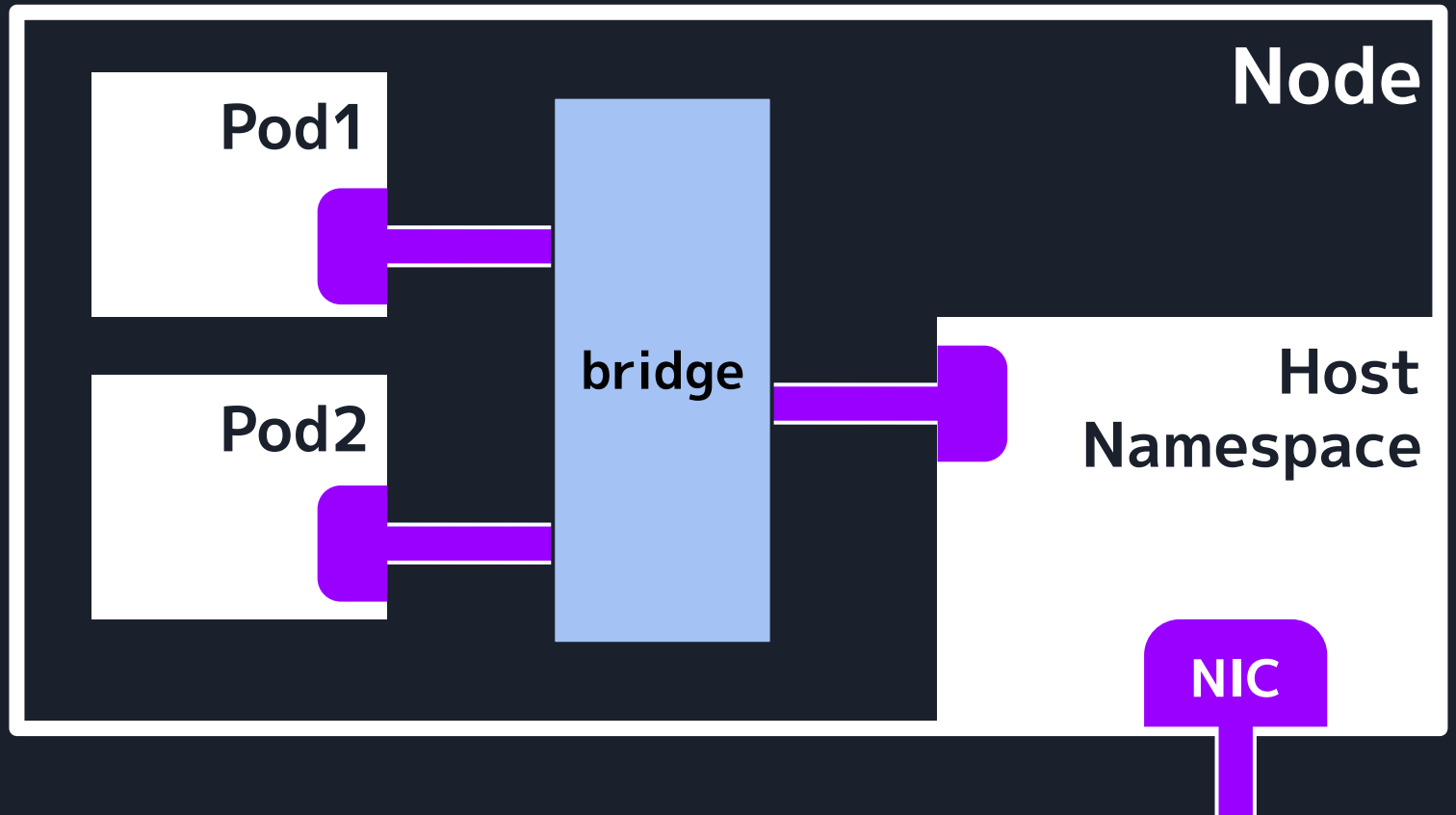


Linux Bridge

Linux Bridge



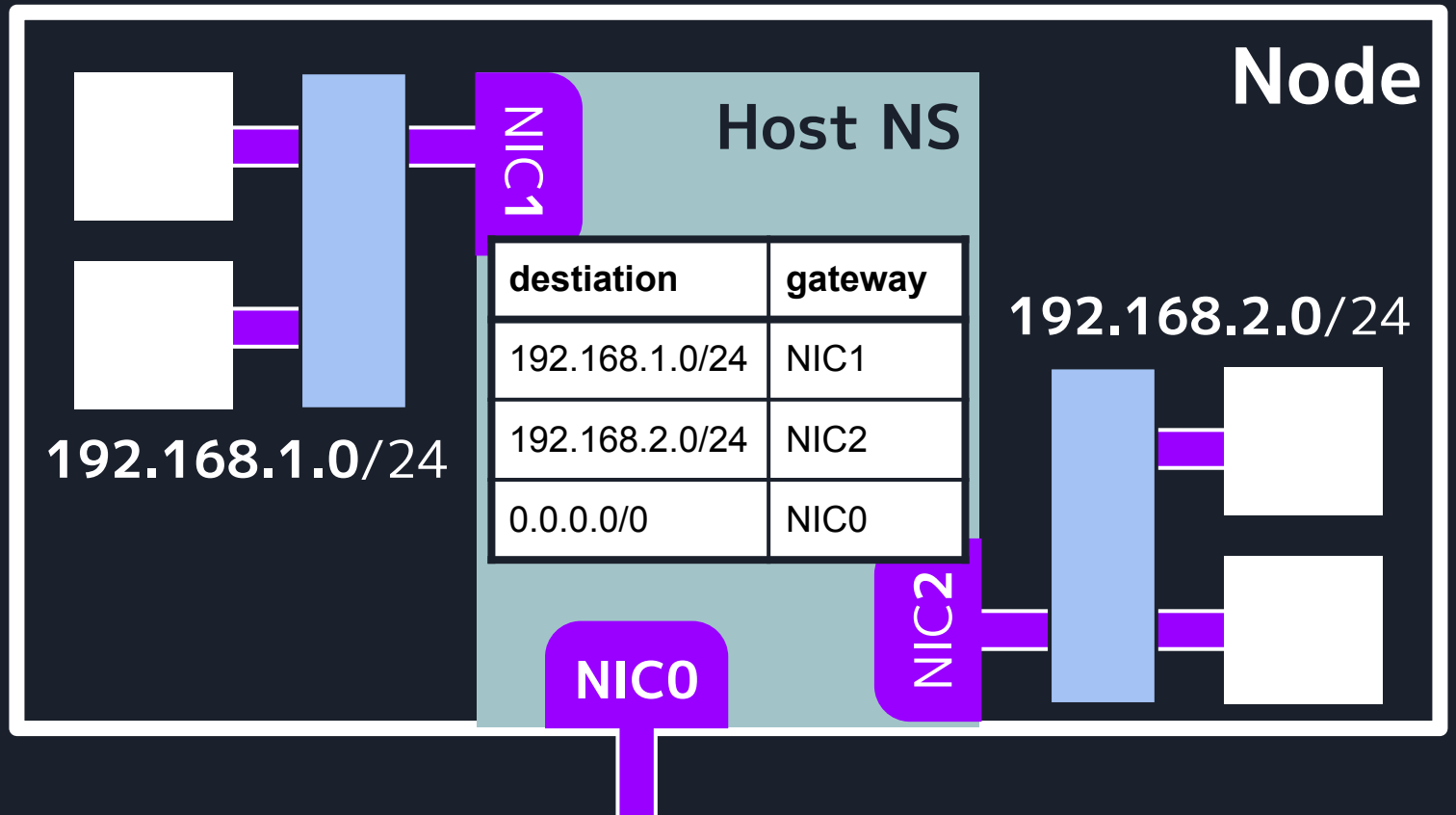
Linux Bridge



ここまでの機能を使うと
1台の親マシンの中で**複雑な**
サブネット構造も作れる

たくさんのルーターが複雑に絡んだ構成も再現可能

ex) 複雑なサブネット構造



netfilter

- パケットを**フィルタ・加工・転送**できる仕組み
  - **Filter**
    - 特定のIPアドレス・ポート宛の**パケットを落とす**
  - **NAT**
    - 送り元・送り先の**IPアドレス・ポートを変更**する
- **操作**方法 (フロントエンド)
  - 長らく**iptables**を使って設定されていたが、**非推奨**に
  - 後継の**nftables**への移行が進んでいる

アジェンダ

- イントロダクション
- Kubernetesを知る
- Kubernetesとネットワーク
- Linuxのネットワークスタック **←イマココ**
- L2/L3ネットワーク: CNI
- L4ロードバランサー: kube-proxy
- プラットフォームとの対話で広がる可能性
- まとめ

アジェンダ

- イントロダクション
- Kubernetesを知る
- Kubernetesとネットワーク
- Linuxのネットワークスタック
- L2/L3ネットワーク: CNI **←イマココ**
- L4ロードバランサー: kube-proxy
- プラットフォームとの対話で広がる可能性
- まとめ

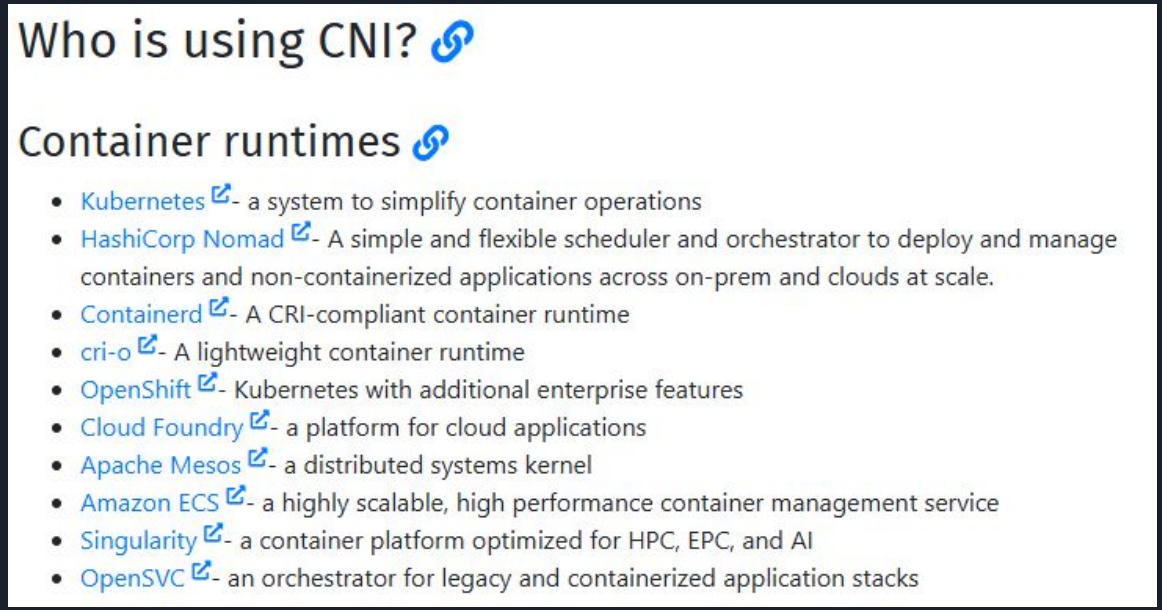
L2/L3ネットワーク:
CNI

CNI

- **Container Network Interface**
- Podを**何らかのネットワークに参加させる**ための仕組み

CNIはKubernetesだけじゃない！

- 意外と多くのコンテナ基盤がCNIに**ネットワーク機能を移譲**している



Who is using CNI?

Container runtimes

- Kubernetes - a system to simplify container operations
- HashiCorp Nomad - A simple and flexible scheduler and orchestrator to deploy and manage containers and non-containerized applications across on-prem and clouds at scale.
- Containerd - A CRI-compliant container runtime
- cri-o - A lightweight container runtime
- OpenShift - Kubernetes with additional enterprise features
- Cloud Foundry - a platform for cloud applications
- Apache Mesos - a distributed systems kernel
- Amazon ECS - a highly scalable, high performance container management service
- Singularity - a container platform optimized for HPC, EPC, and AI
- OpenSVC - an orchestrator for legacy and containerized application stacks

CNIはKubernetesだけじゃない！

- 意外と多くのコンテ
  移譲している

Containerd (ただの**コンテナランタイム**)からも**CNIは使える**！

Who is using CNI?

Container runtimes

- Kubernetes - a system to simp operations
- HashiCorp Nomad - A flexible scheduler and orchestrator to deploy and manage containers and non-containerized applications across on-prem and clouds at scale.
- Containerd - A CRI-compliant container runtime
- cri-o - A lightweight container runtime
- OpenShift - Kubernetes with additional enterprise features
- Cloud Foundry - a platform for cloud applications
- Apache Mesos - a distributed systems kernel
- Amazon ECS - a highly scalable, high performance container management service
- Singularity - a container platform optimized for HPC, EPC, and AI
- OpenSVC - an orchestrator for legacy and containerized application stacks

余談: CNIは (実質) Kubernetesだけ

- DockerやPodmanなど、多くの人がローカルで使うコンテナランタイムは、独自の**ドライバ**を使って管理する
  - Containerd (Dockerの中身) を直接触る人は少ない
- 実際のユースケースは、ほぼ**Kubernetesか類似のオーケストレーションサービス**
- Kubernetesのネットワークのメンテナーも「CNIはK8sに**特化した仕組みじゃない**のに、ほぼK8sから**しか使われない**」と言っている

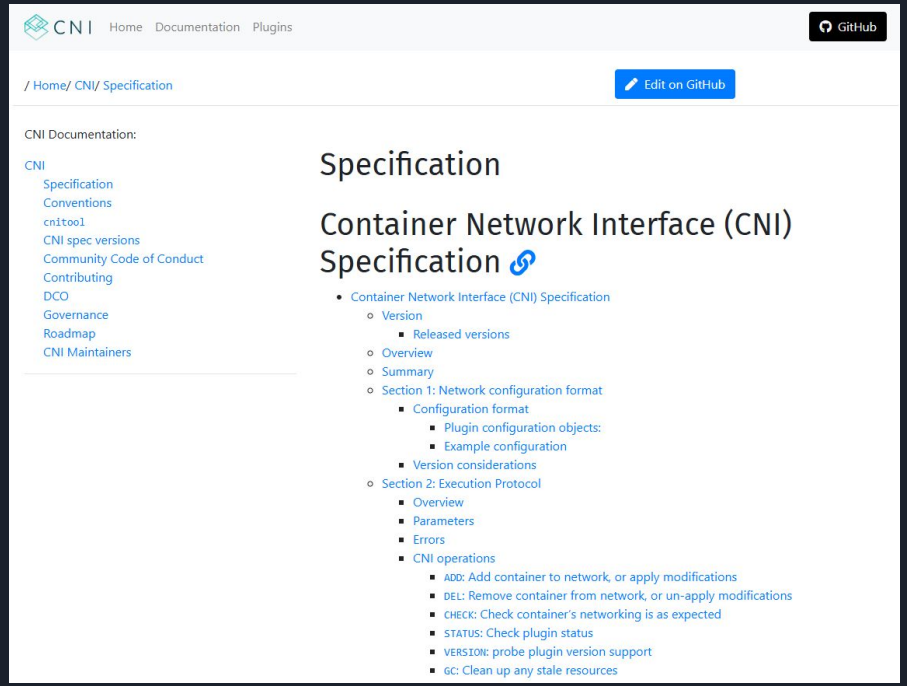
広義「CNI」とは

- CNI**仕様** (containernetworkinterface/cni)
  - コンテナ基盤が**どのようにプラグインを呼び出すか**の定義
- 基盤の**要件定義**
  - 基盤として**どんなネットワークが欲しい**のか
- CNI**プラグイン**
  - CNI仕様を通して操作が可能で、基盤の要件定義を満たす**実際のネットワーク実装**

CNI仕様

CNI仕様

- プラグインが満たすべき**ユーザーとのインターフェース**が定められている
  - **設定**方法
  - **操作**方法
  - 複数CNIの連携



CNI仕様 - 操作コマンド

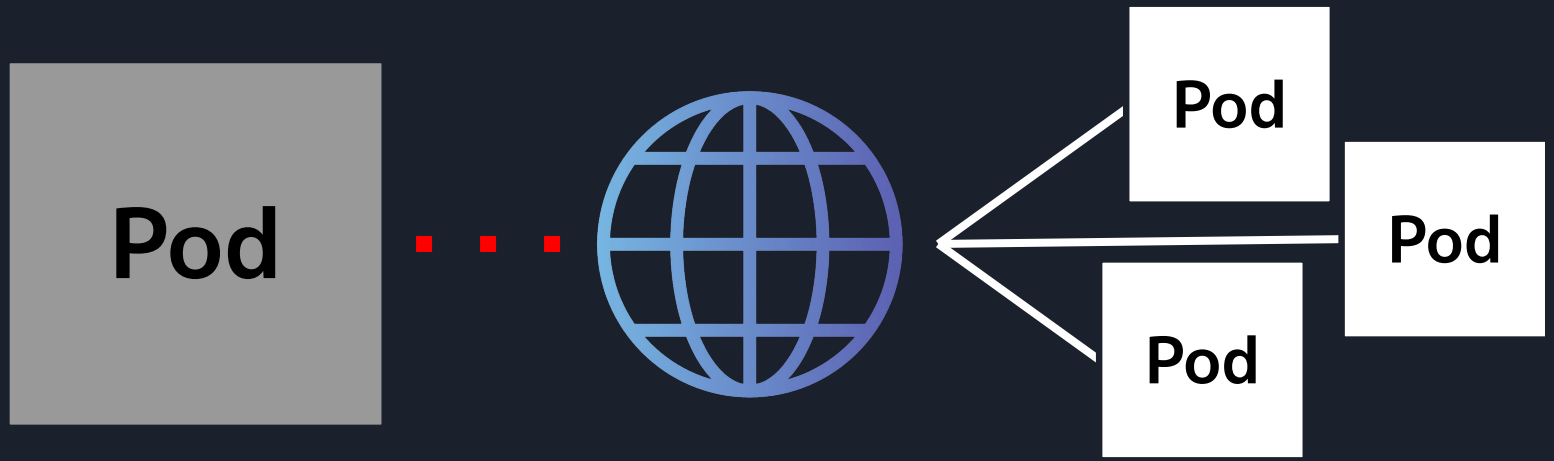
- CNIプラグインは**5つのサブコマンド**を備えた**バイナリ**として実装される必要がある
  - ADDコマンド
  - DELコマンド
  - CHECKコマンド - 現在の状態を確認
  - GCコマンド - いらないリソースを掃除
  - VERSIONコマンド
- 特に大事なのが**ADD**と**DEL**

CNI仕様 - 操作コマンド

- **ADD**コマンド
  - コンテナ (Pod) に**IPアドレス**を割り当てる
  - 指定されたコンテナを、用意された**コンテナネットワークの中に参加**させる

CNI仕様 - 操作コマンド

- **DEL**コマンド
  - 指定されたコンテナ (Pod) を**ネットワークから削除**する



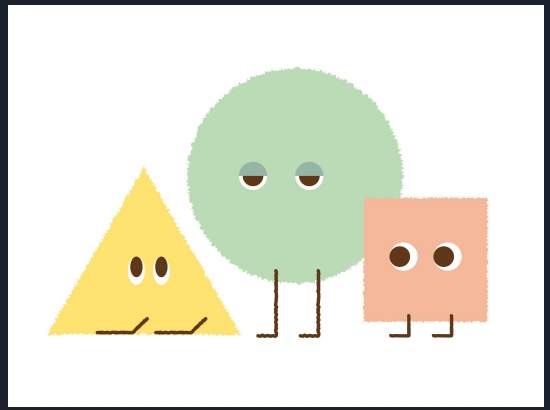
CNI仕様

- 逆に言えば、この2つのコマンドが処理できればCNIプラグインとして動く
  - **Bash**でCNIプラグインを**作った**人もいます
- Kubernetesの場合CNIのコア機能を**デーモンで作動**させ**エントリーポイント**になるバイナリを別途用意し、そこからデーモンのAPIを叩きにいく構成が多い
  - 複雑なネットワーク**ライフサイクル**に対処するため

基盤の**要件定義**

基盤側の要件定義

- コンテナ基盤によって**必要な要件は違う**
- 例: **ローカル**環境
  - 同一ホスト内でコンテナ同士の疎通が取れる
  - コンテナから外部ネットワークに出られる



The Kubernetes network model

- 各Podは**クラスタ内で唯一**のIPアドレスを持つ
- ホストの同じ/異なるに**関わらず**、**NAT無し**でPod同士の通信ができる

https://kubernetes.io/docs/concepts/services-networking/

The Kubernetes network model

すなわち、CNIが**やらなければいけない**ことは

- Podにユニークな**IPアドレスを割り当て**る (IPAM)
- 同じクラスターのPod同士が、ホストの同じ/異なるに**関わらず**、**NATなし**で通信できるようにする

CNIという仕様が担保する「自由」

- 逆に、これらが実現できれば使う手法は**なんでも良い**
  - これからいくつか例を見ていくが、**千差万別**
- CNI仕様は**どんなネットワーク技術でも受け入れる**広い受け皿
  - クラウド事業者が**元々のネットワーク基盤を活用**しやすいなど、様々なメリットがある

CNIプラグインとその実装

メジャーなOSSプラグイン

- **Flannel**
  - ノードを跨ぐPod間通信に**VXLAN**を使用したCNI
- **Calico**
  - 各ノードの持つPod CIDRを**BGP**で広報するCNI
- **Cilium**
  - ノード内の通信に**eBPF**を活用したCNI

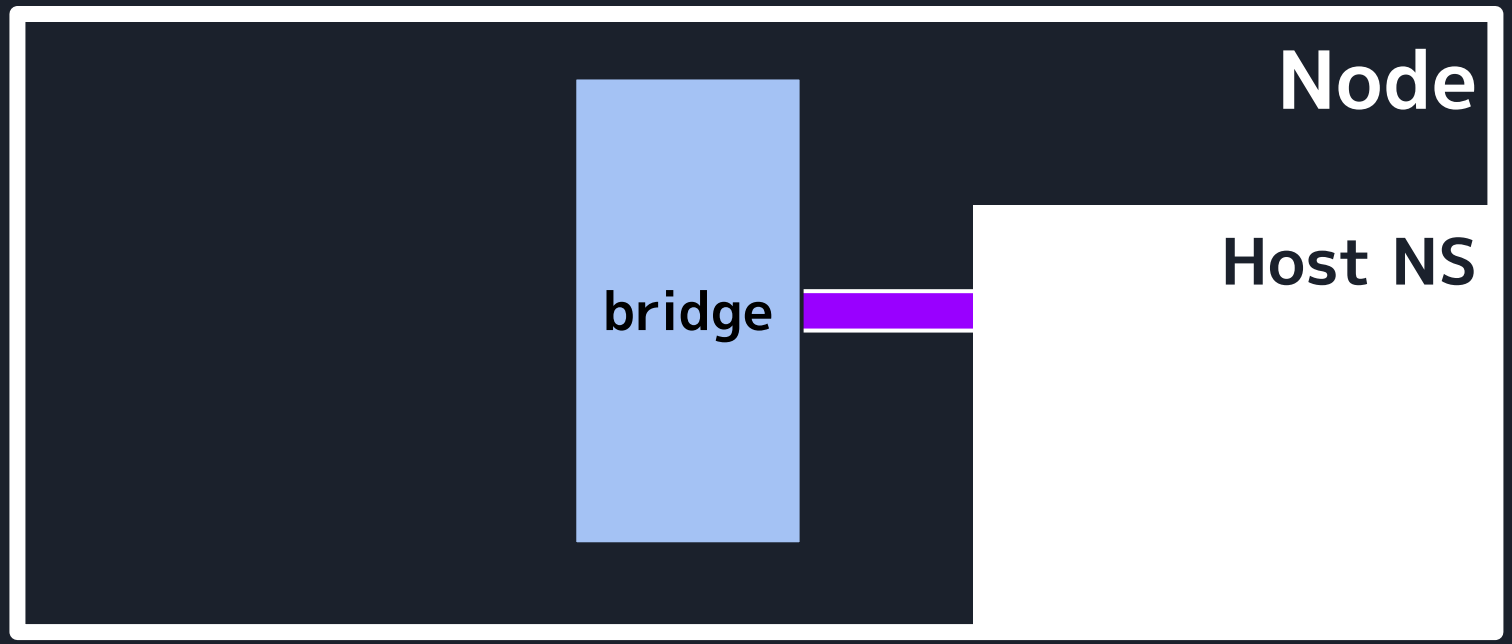
この3つを例に、それぞれの**デフォルトの通信方式**がどのように疎通性を実現しているか見ていきましょう

Flannel - 同じノード内

下準備

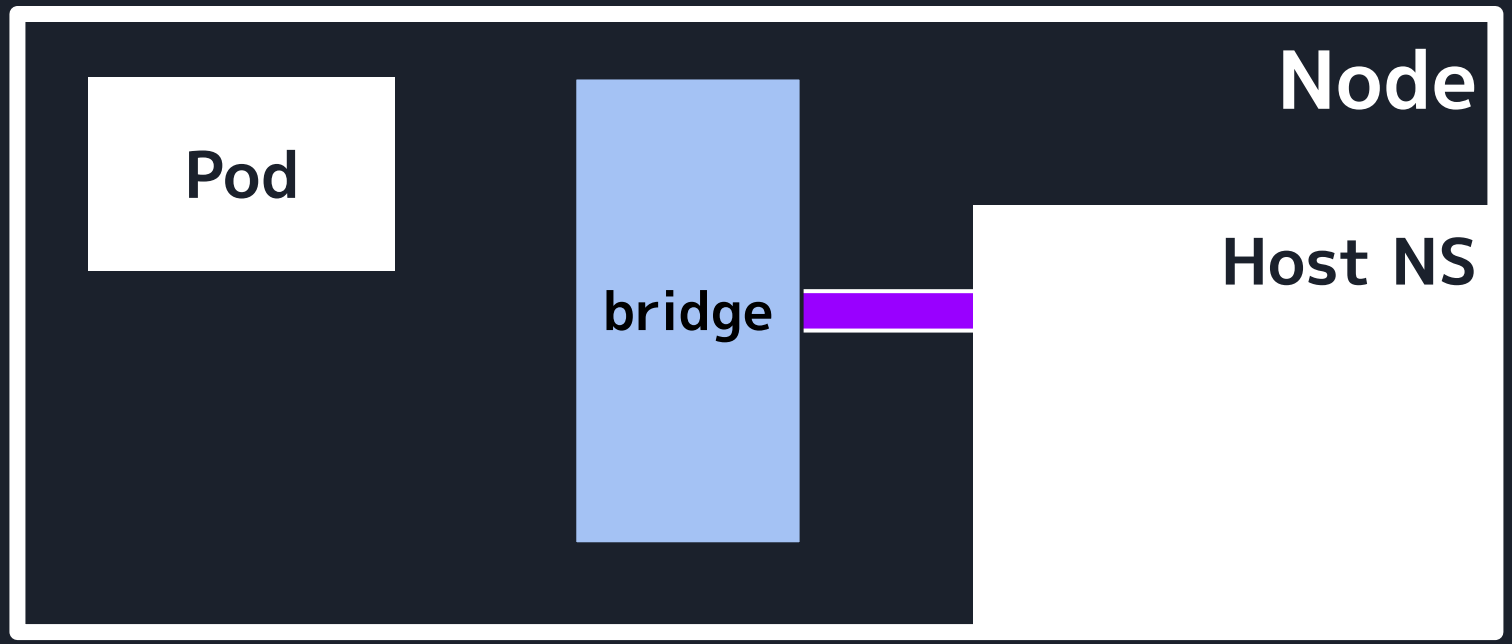
Flannel - 同じノード内

下準備: **Linux Bridge**を作る



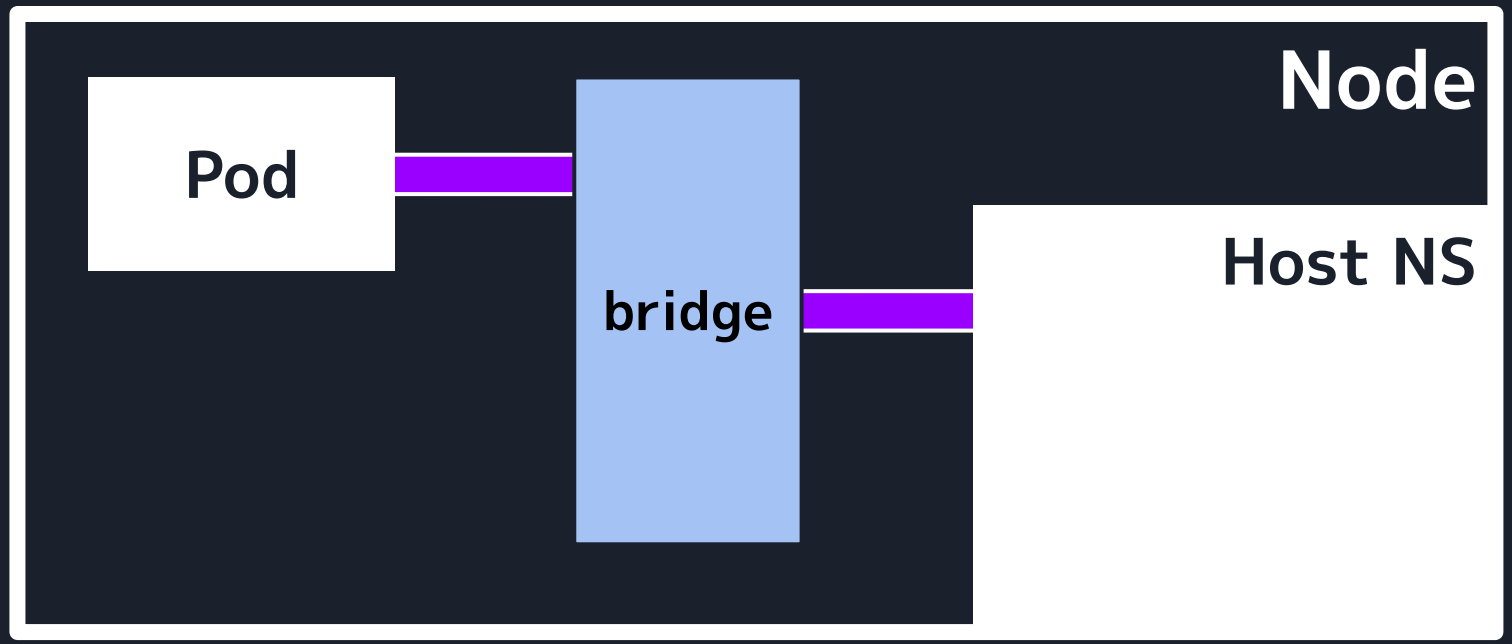
Flannel - 同じノード内

Pod作成時



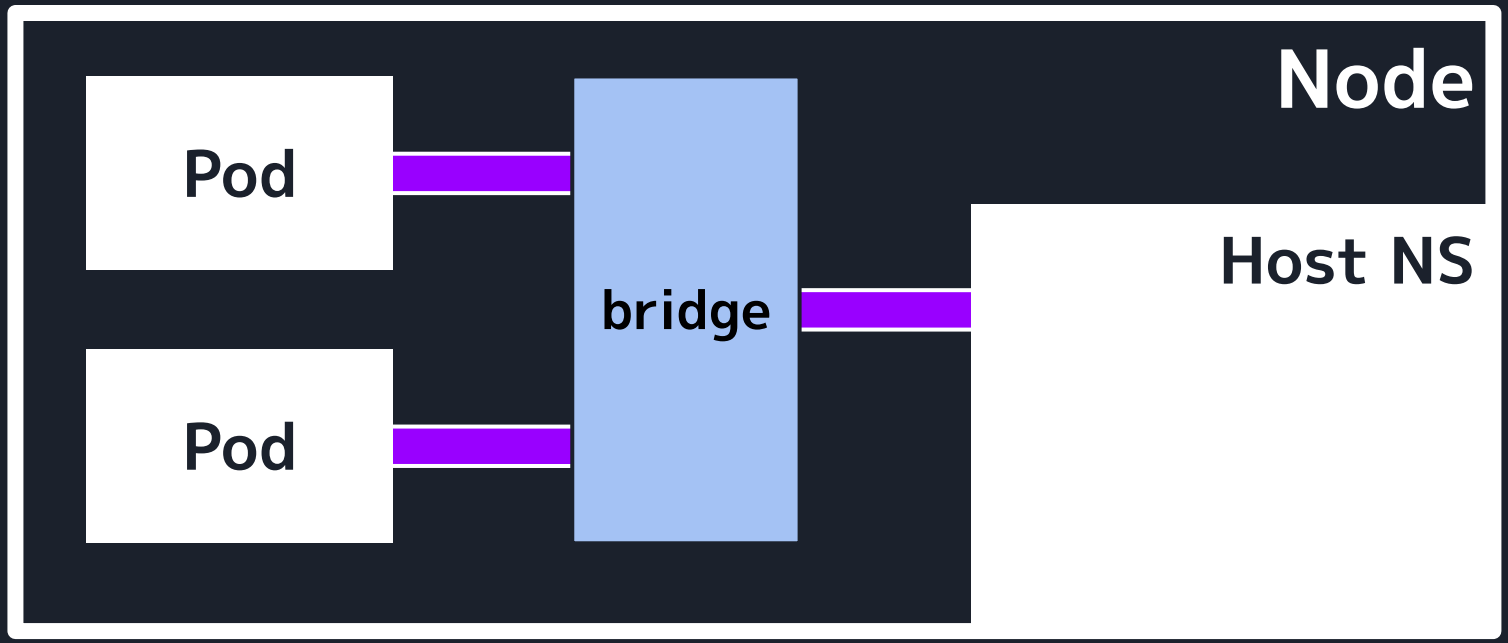
Flannel - 同じノード内

Pod作成時: Linux Bridgeに**接続**



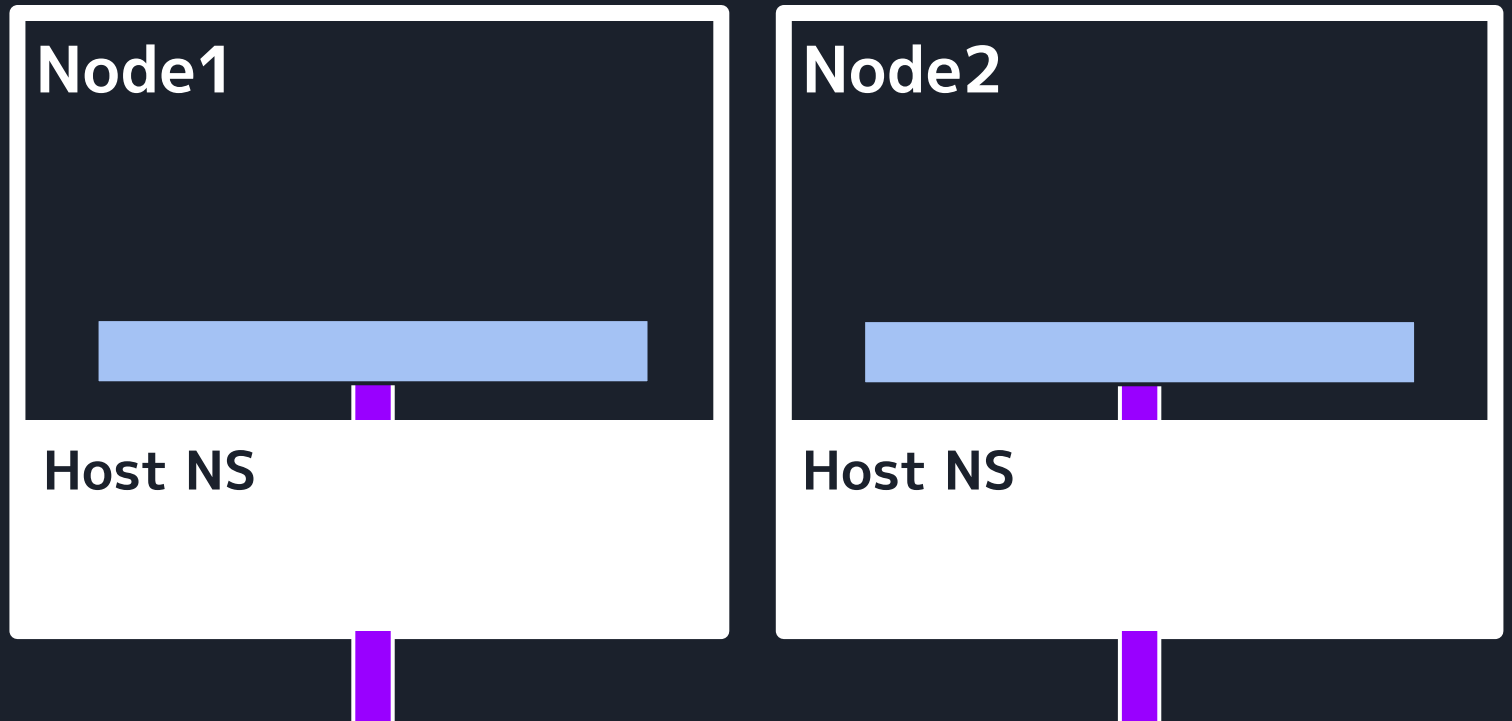
Flannel - 同じノード内

通信時: 全てのノード内Podは**L2で接続**され、疎通ができる



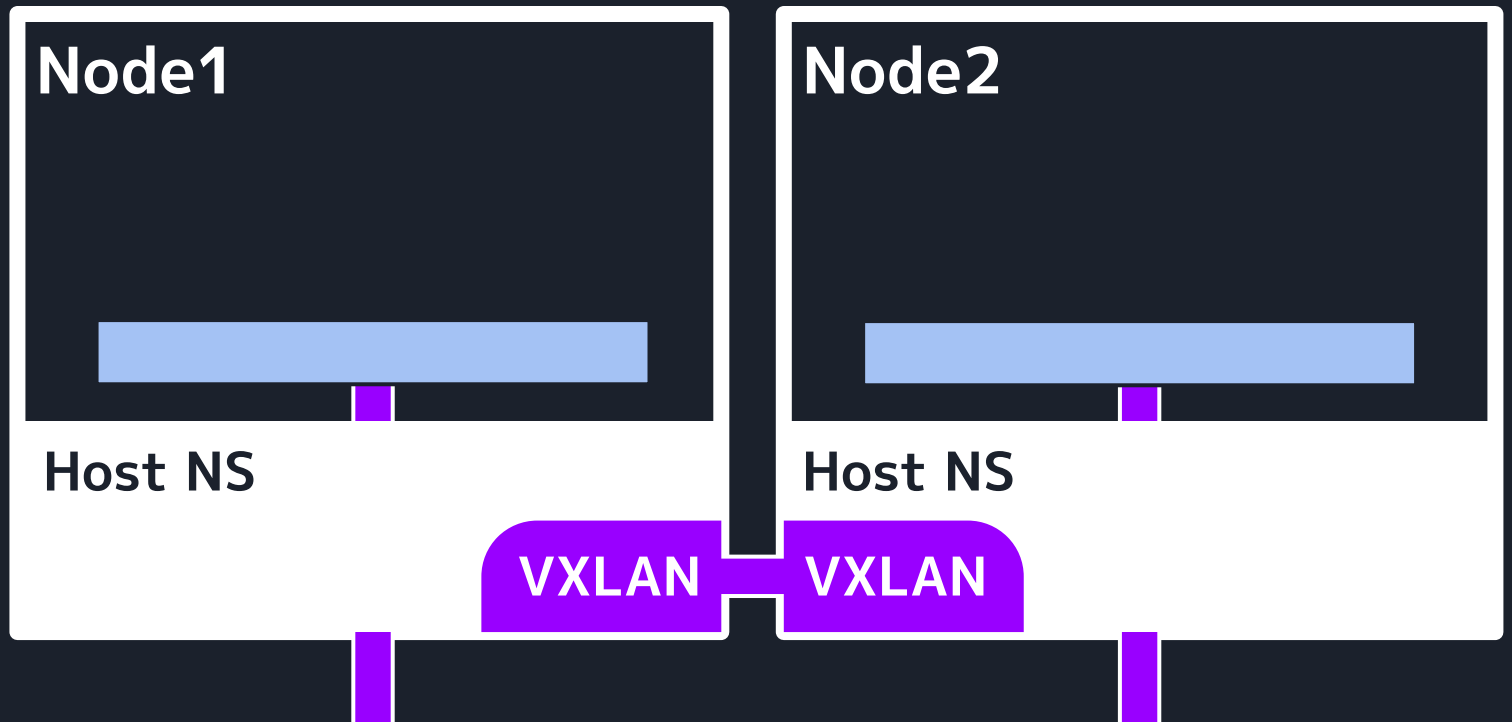
Flannel - 異なるノード間

下準備: 同じノード内の下準備が終わった段階



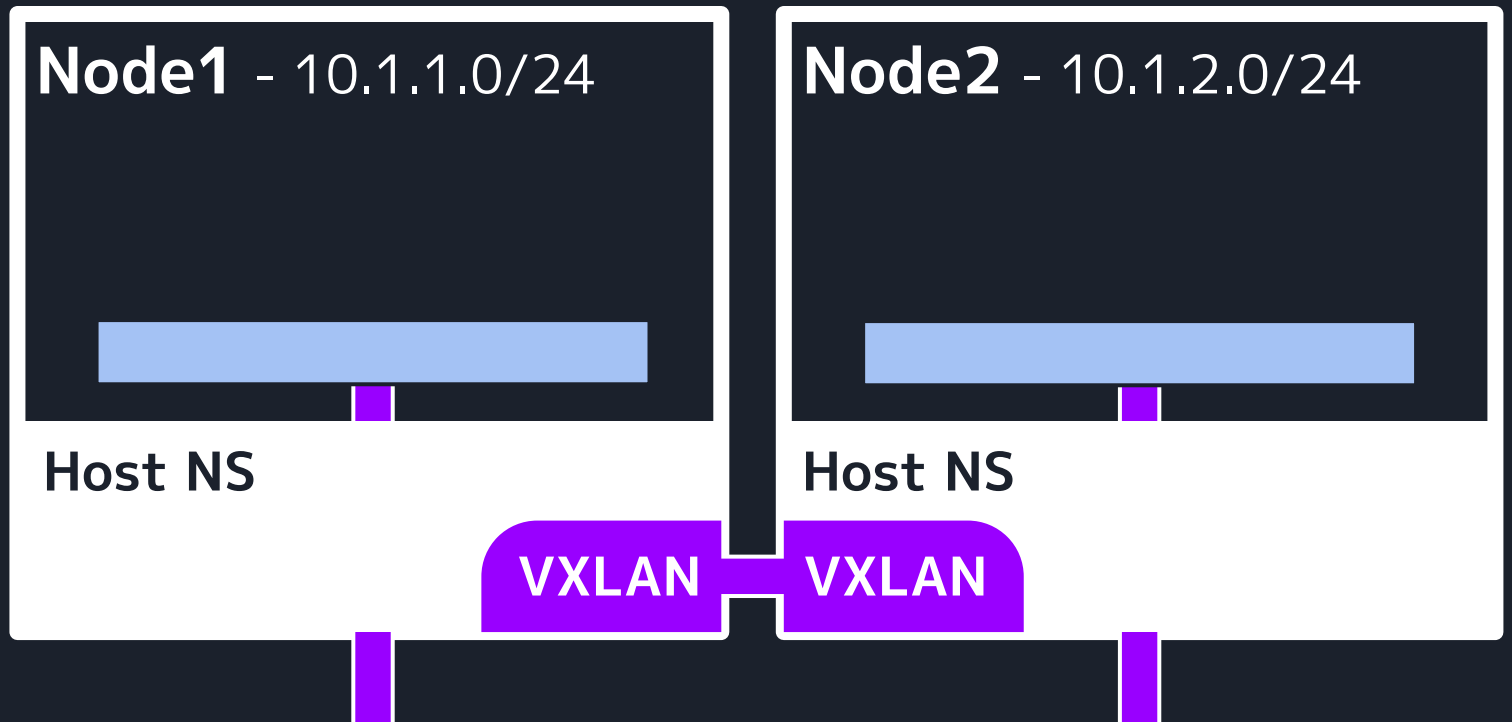
Flannel - 異なるノード間

下準備: 各ノードで**VTEP**を作り、**VXLAN**で接続



Flannel - 異なるノード間

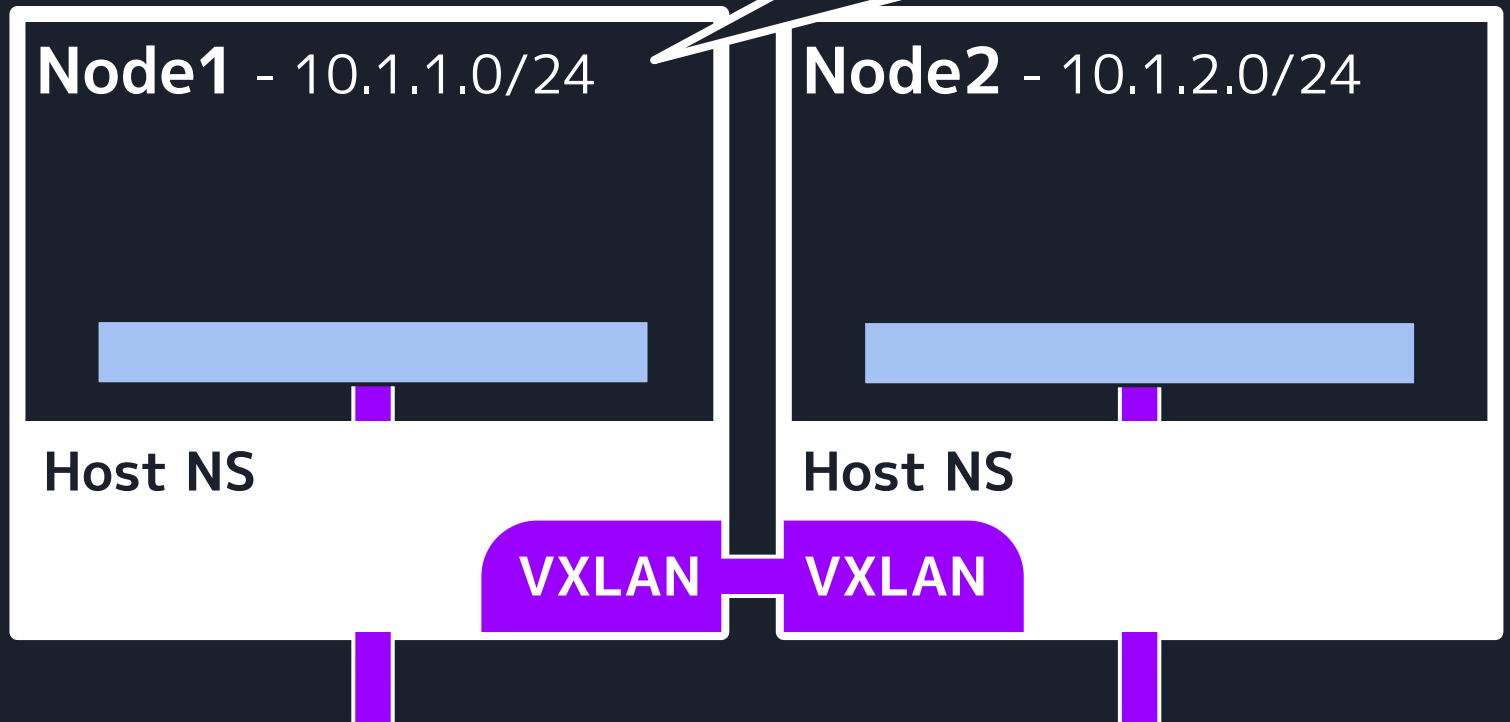
下準備: 各ノードに**固有のサブネット**を割り当てる



Flannel - 異なるノ

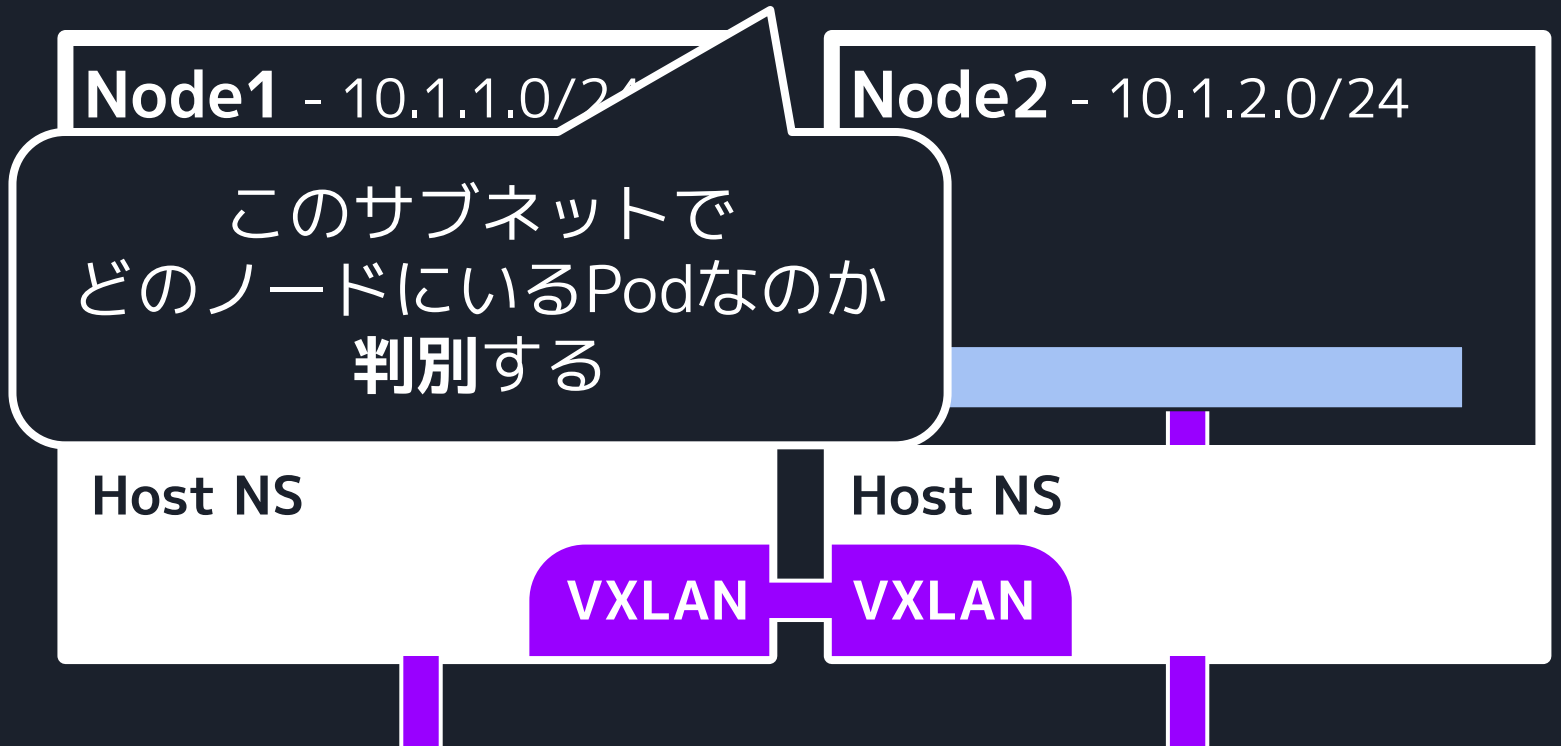
Pod全体のIPレンジは
10.1.0.0/16とする

下準備: 各ノードに**固有のサ**



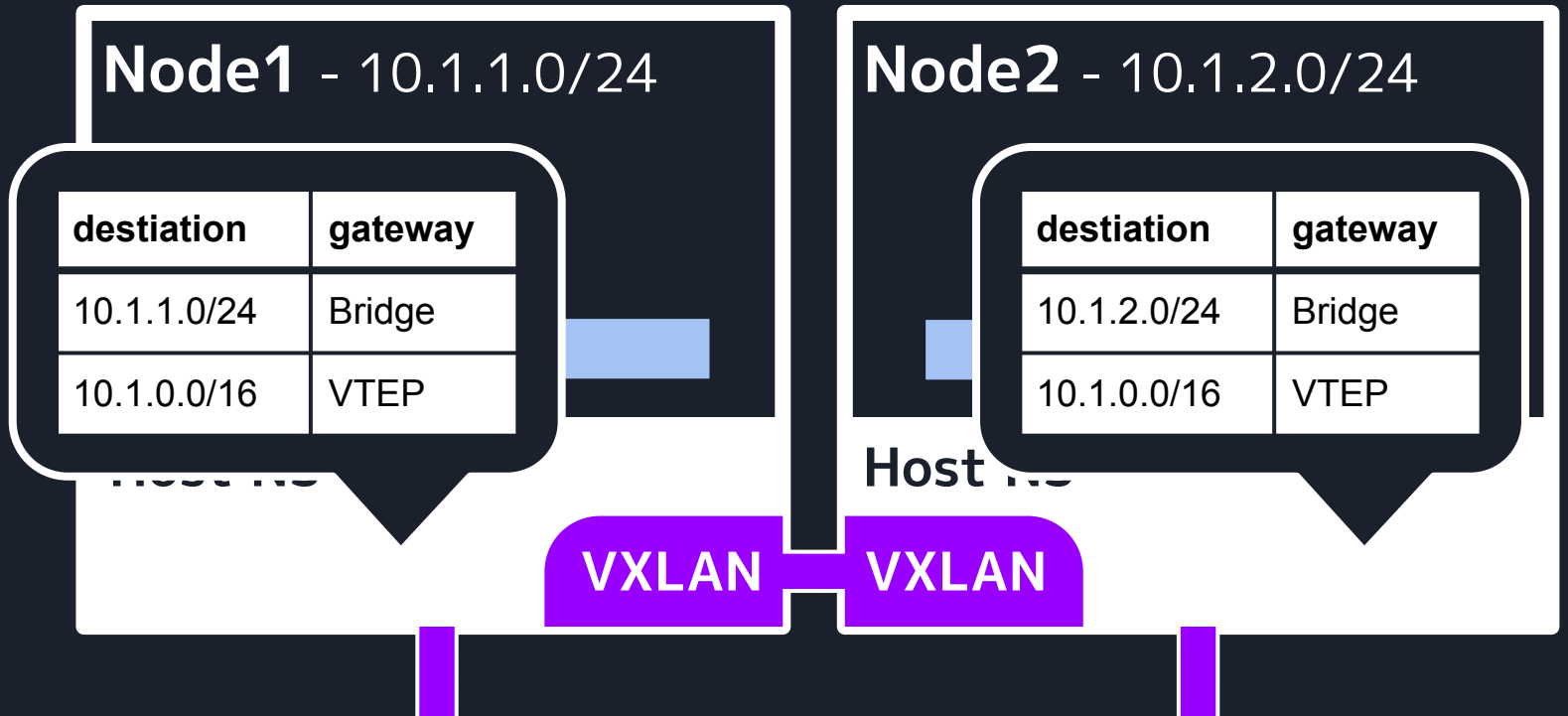
Flannel - 異なるノード間

下準備: 各ノードに**固有のサブネット**を割り当てる



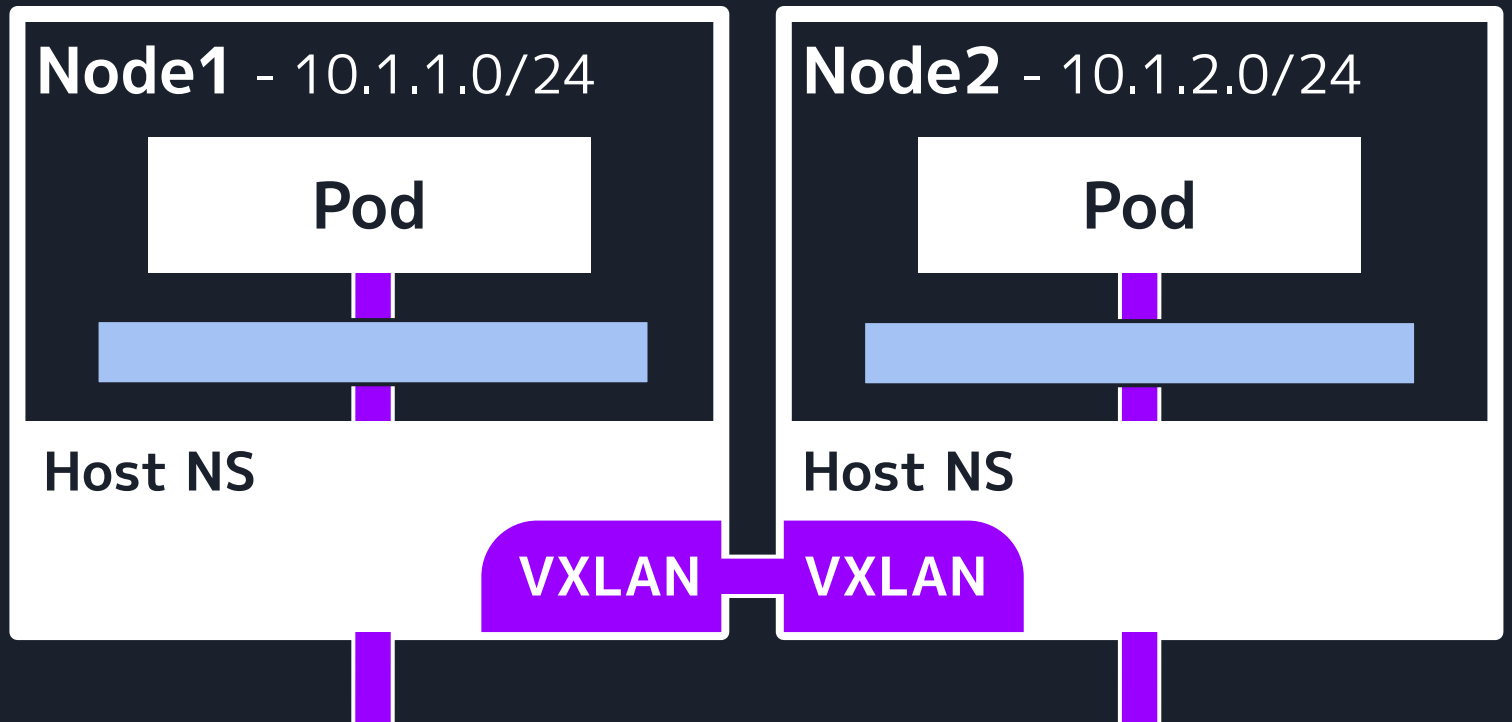
Flannel - 異なるノード間

下準備: **ルーティングテーブル**にエントリを追加



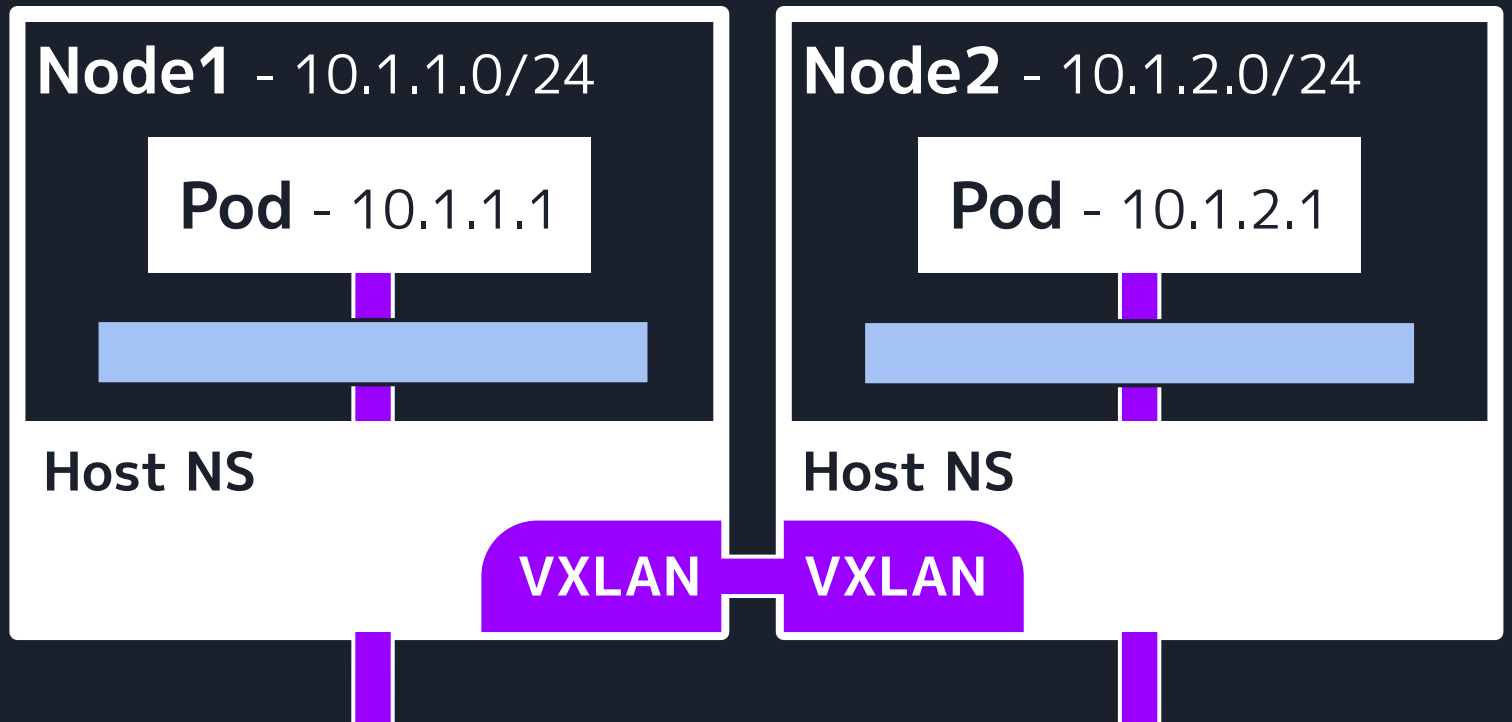
Flannel - 異なるノード間

Pod作成時



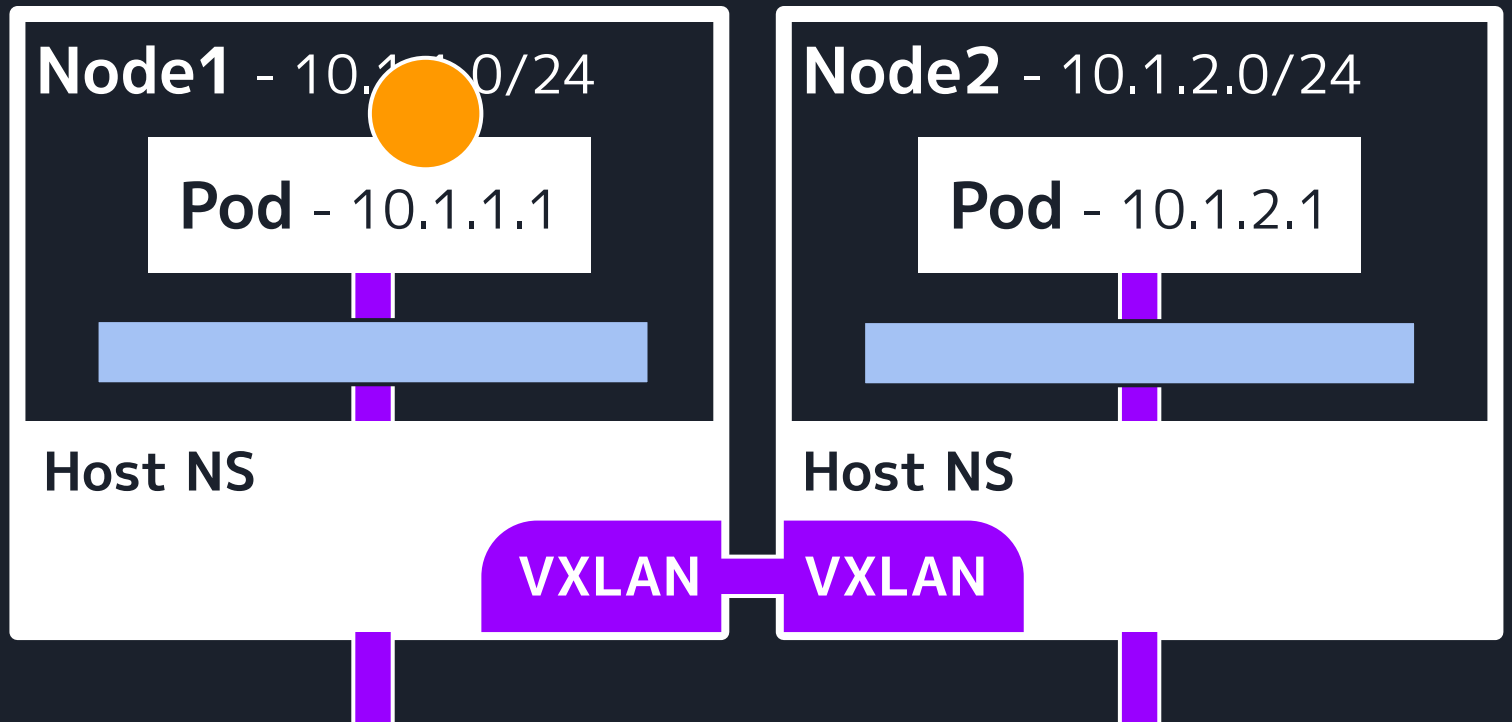
Flannel - 異なるノード間

Pod作成時: ノードのIPレンジから**アドレスを割り当て**る



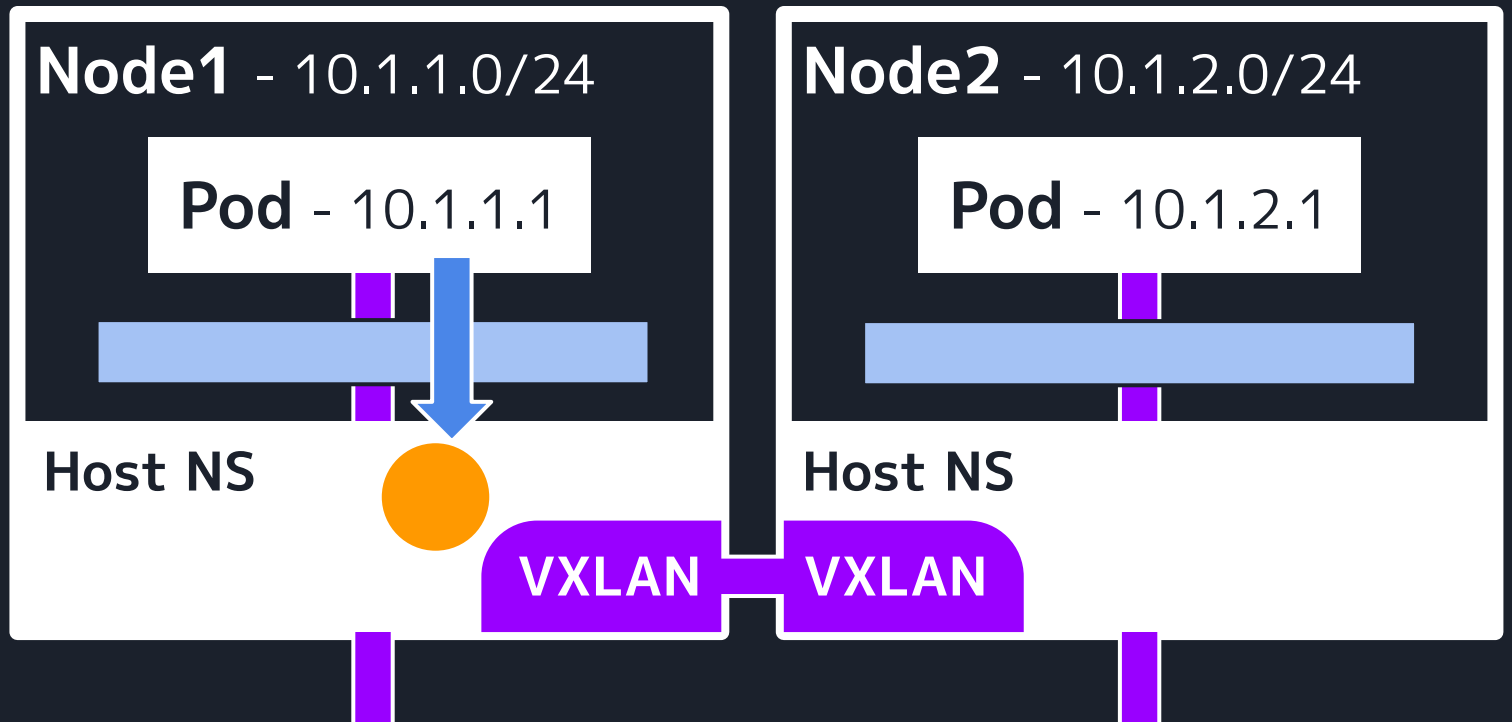
Flannel - 異なるノード間

通信時: **10.1.1.1**から**10.1.2.1**へ送る時



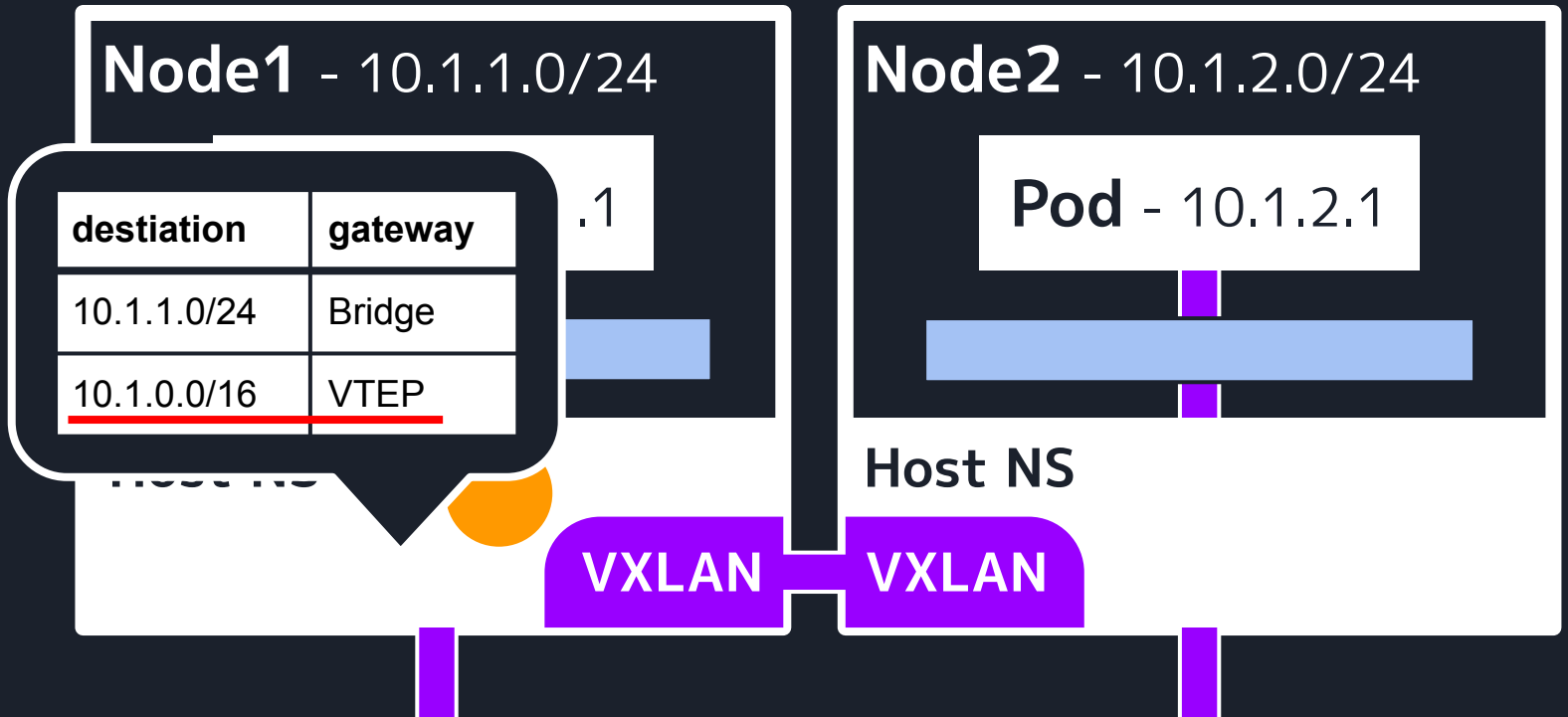
Flannel - 異なるノード間

通信時: **デフォルトゲートウェイ**となっているHost NSへ



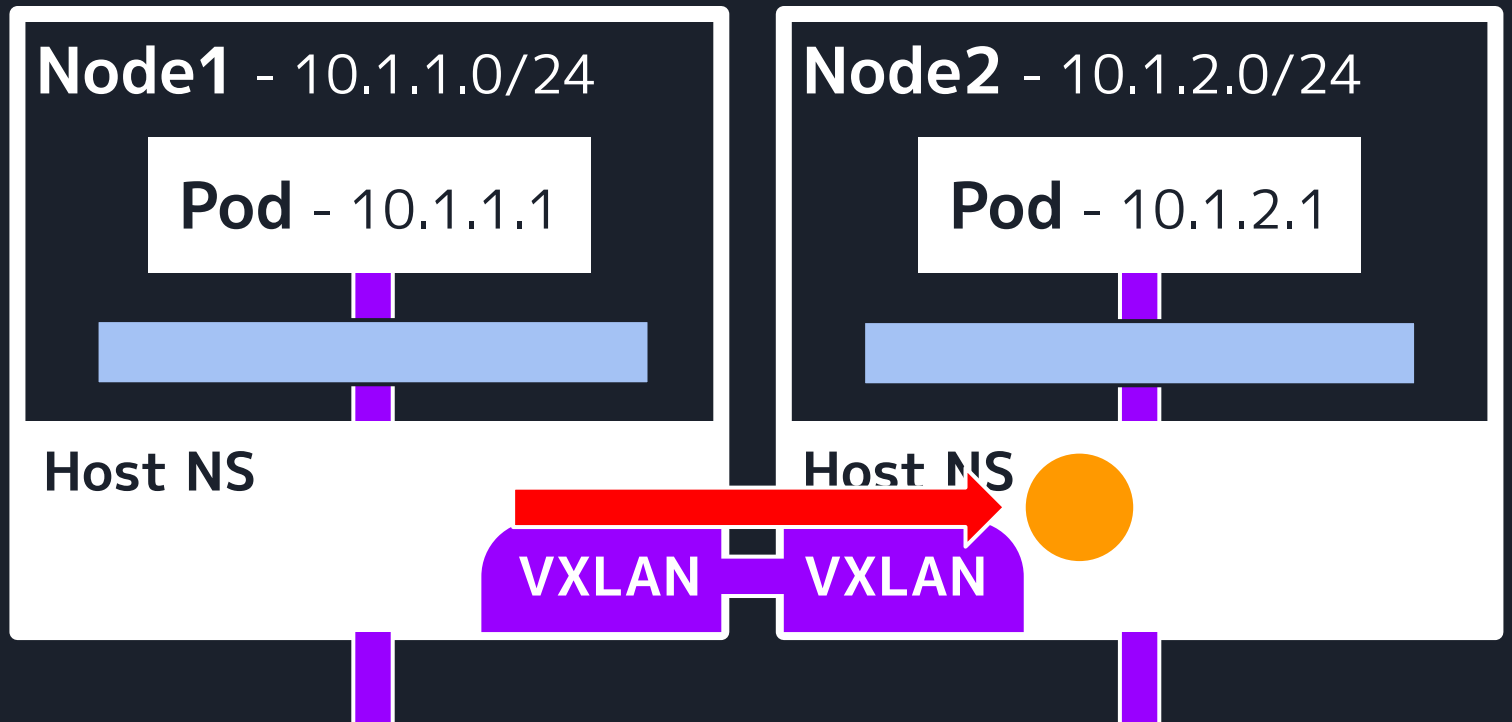
Flannel - 異なるノード間

通信時: **ルーティングテーブル**を確認



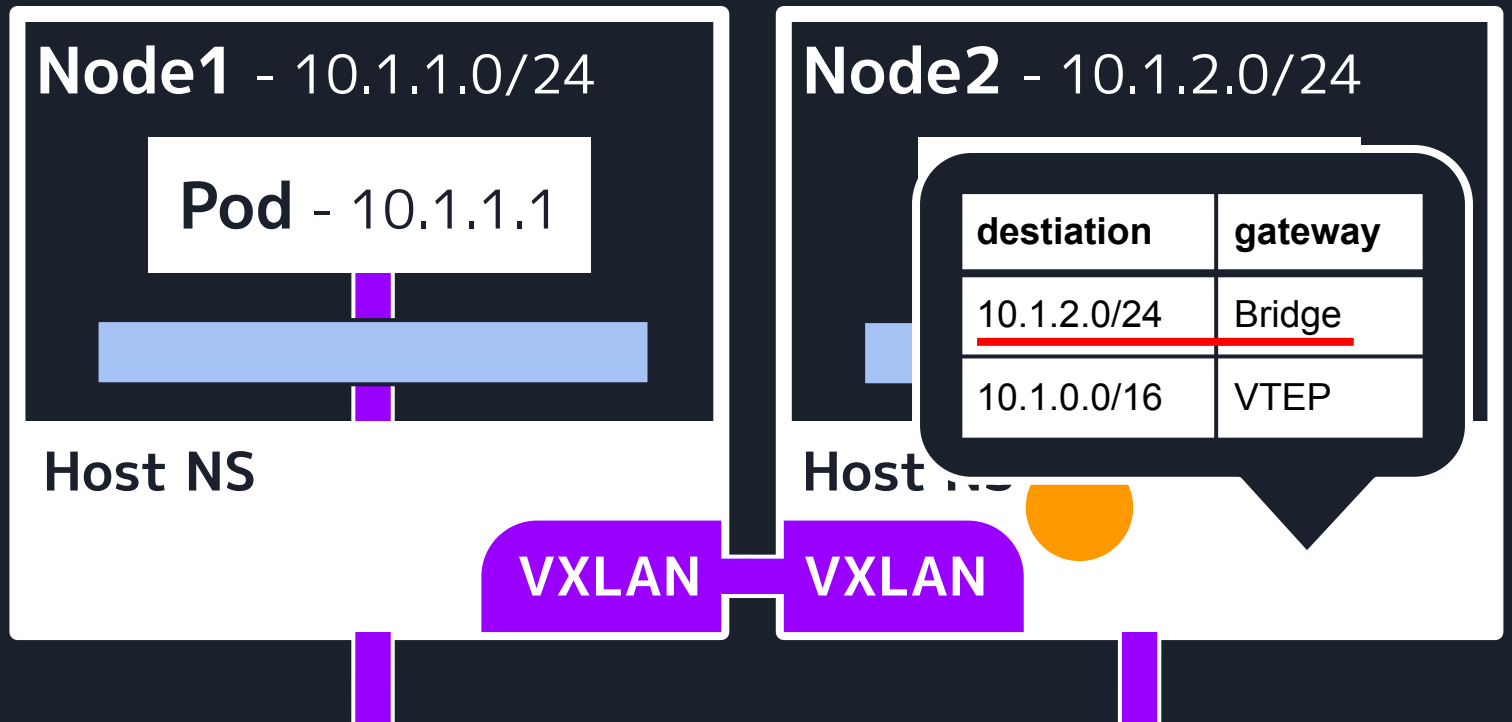
Flannel - 異なるノード間

通信時: gatewayの**VTEP**から**VXLAN**経由でNode2へ送信



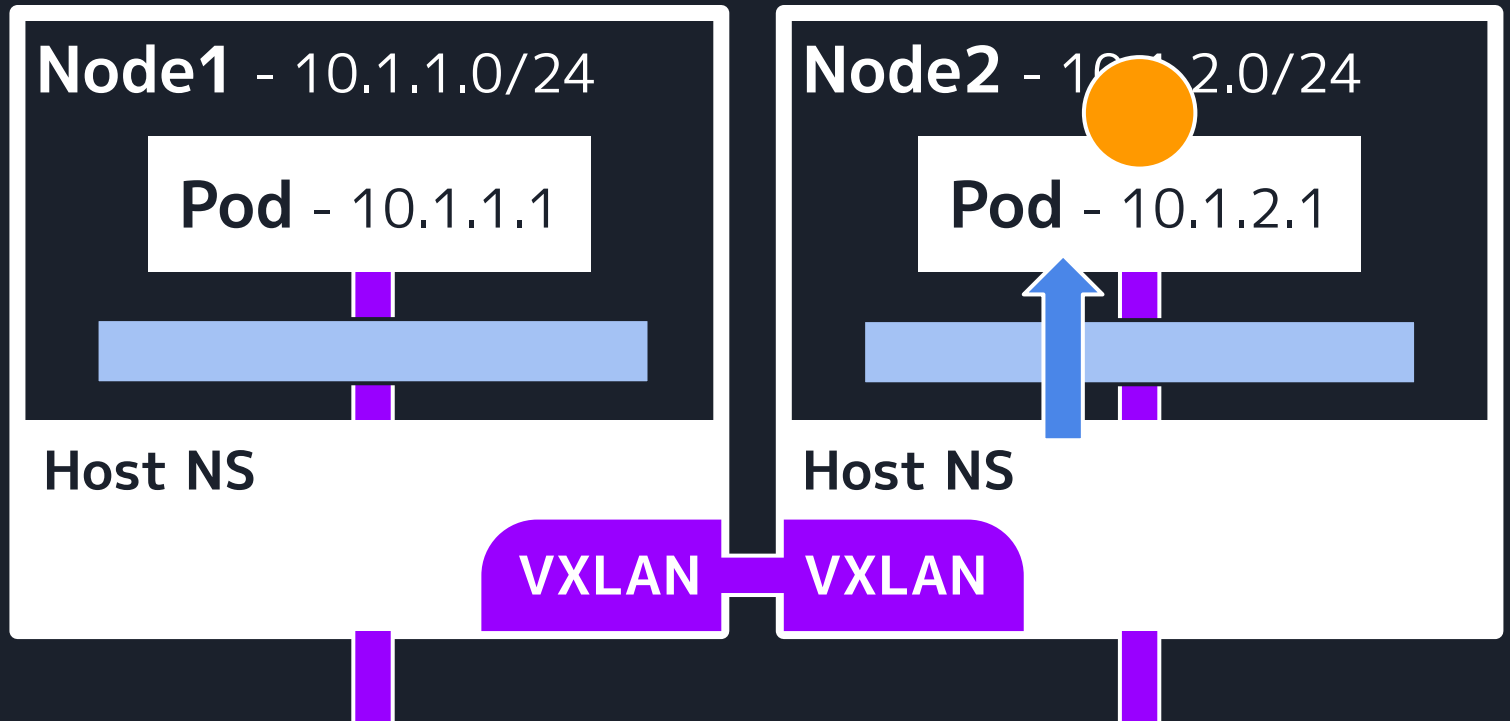
Flannel - 異なるノード間

通信時: **ルーティングテーブル**を確認



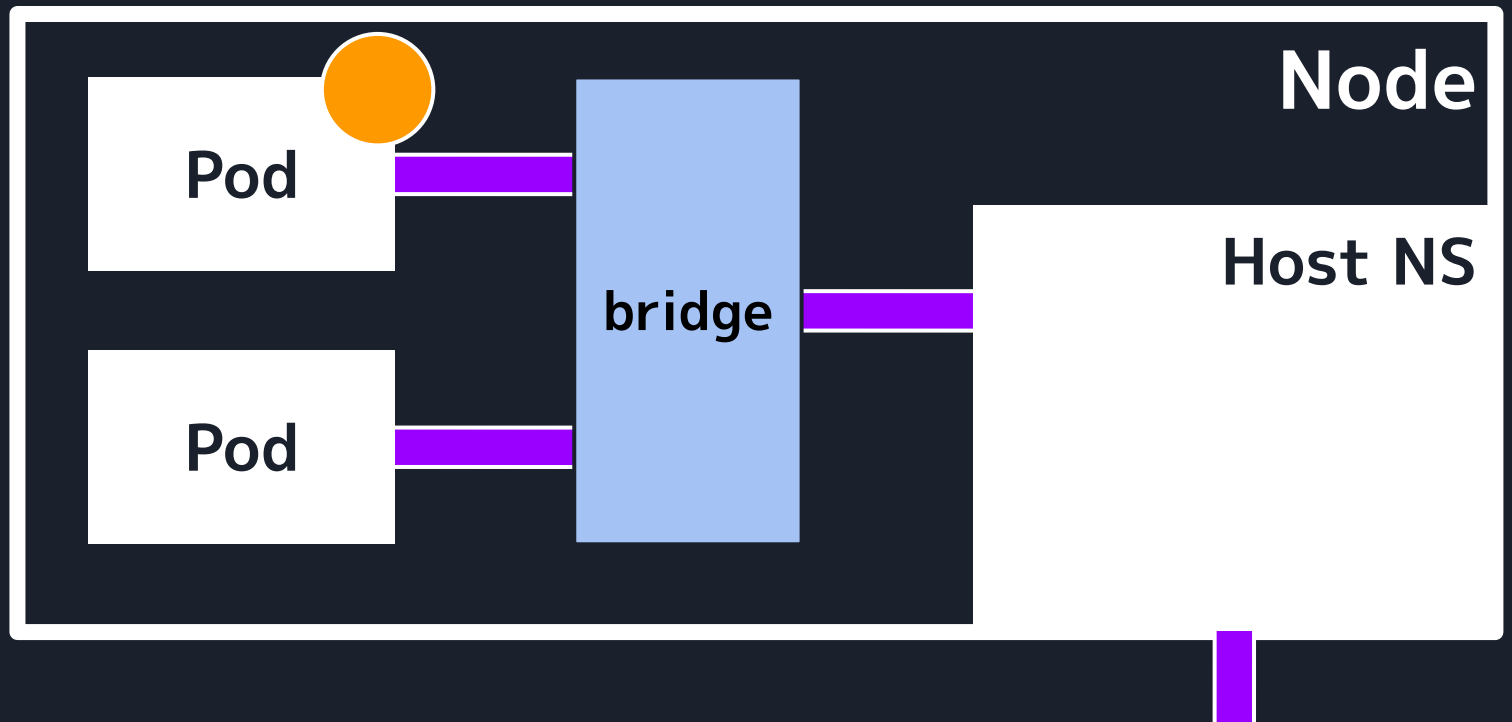
Flannel - 異なるノード間

通信時: gatewayの**Bridge**を経由し、宛先に到達



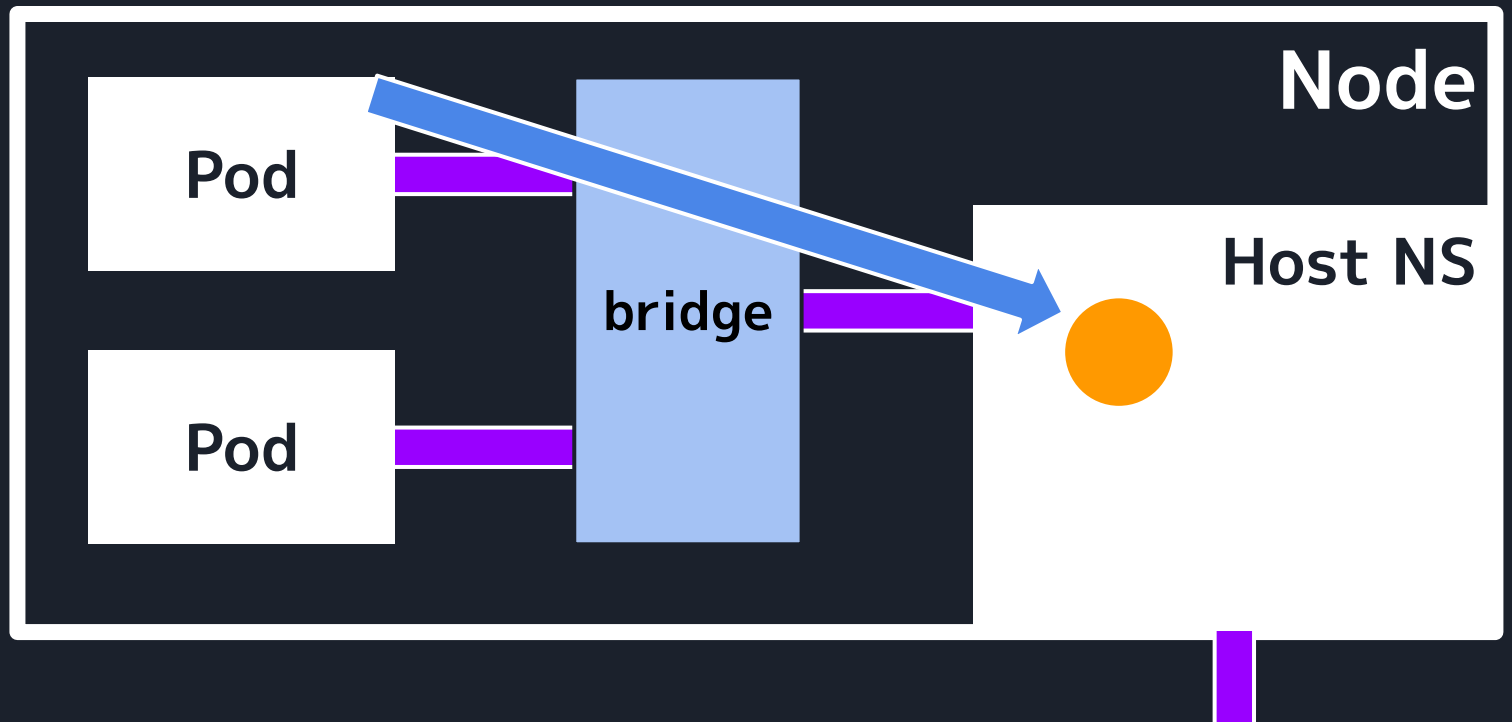
Flannel - Podから外部ネットワーク

NetfilterでIPマスカレードをして外部に出す



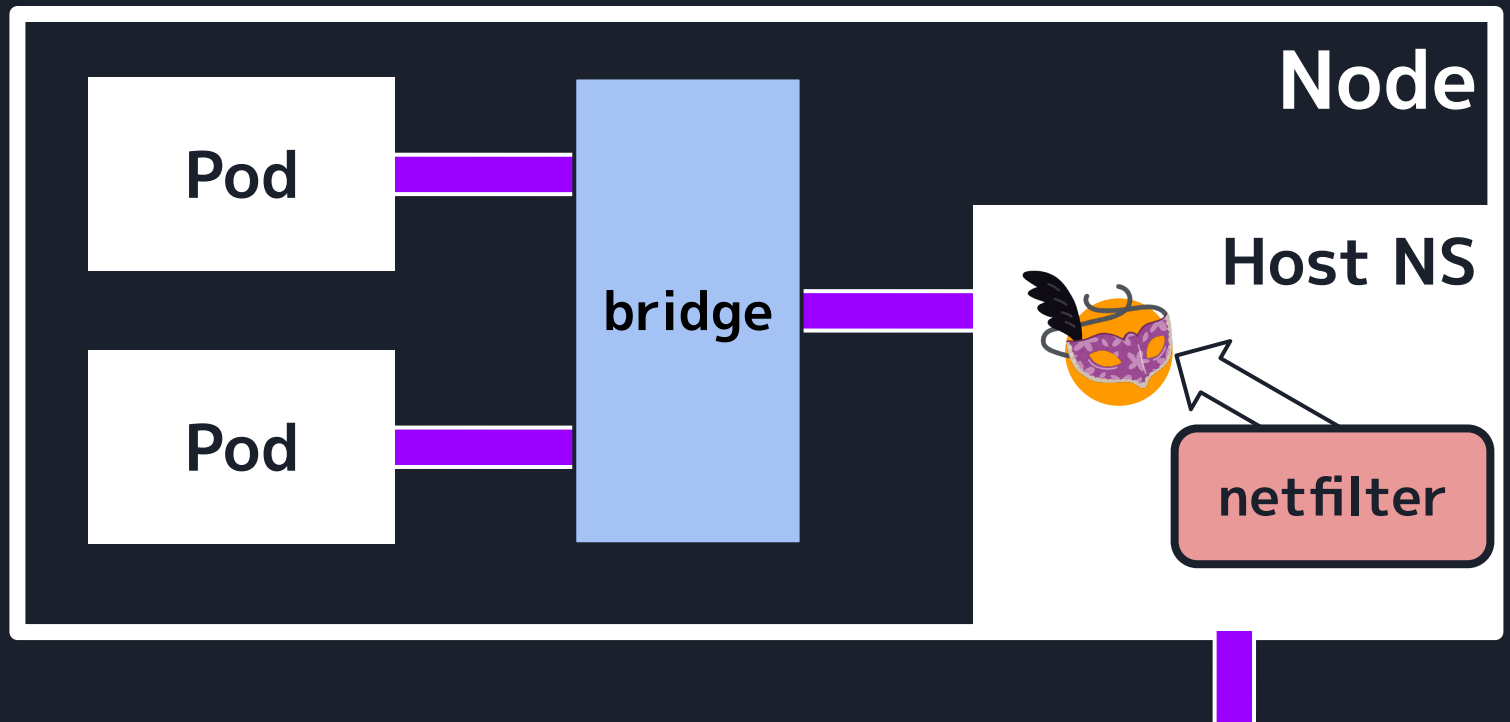
Flannel - Podから外部ネットワーク

NetfilterでIPマスカレードをして外部に出す



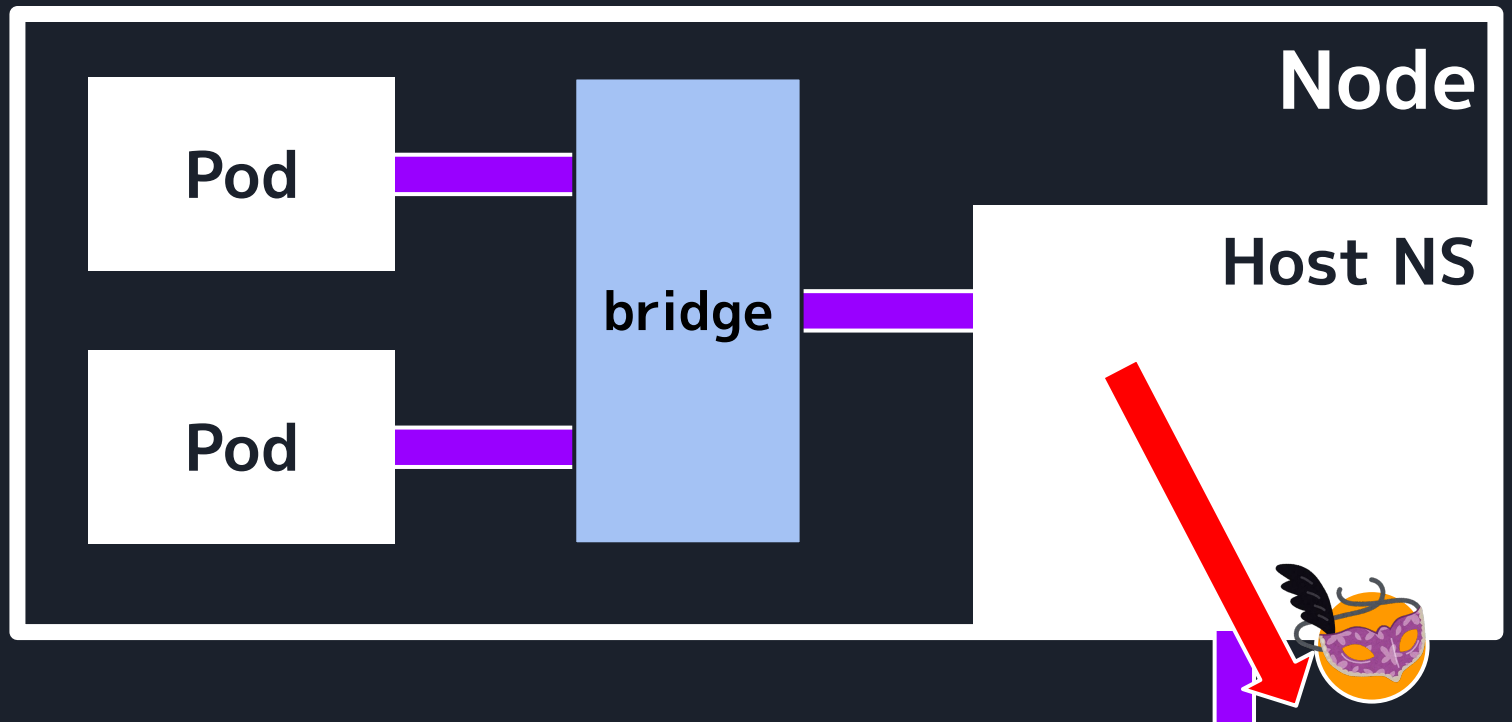
Flannel - Podから外部ネットワーク

NetfilterでIPマスカレードをして外部に出す



Flannel - Podから外部ネットワーク

NetfilterでIPマスカレードをして外部に出す



Flannel - 特徴まとめ

- **同じノード内**でのL3疎通
  - **Linux Bridge**で全てのPodをL2接続する
- **異なるノード間**のL3疎通
  - **VXLAN**を用いて別ノードにパケットを丸ごと輸送する**オーバーレイネットワーク**
    - 余談) Wireguardなど**VPN**で輸送するモードも
- **サブネット**でノードを見分ける
- **netfilter**でIPマスカレードする

Calico - 同じノード内

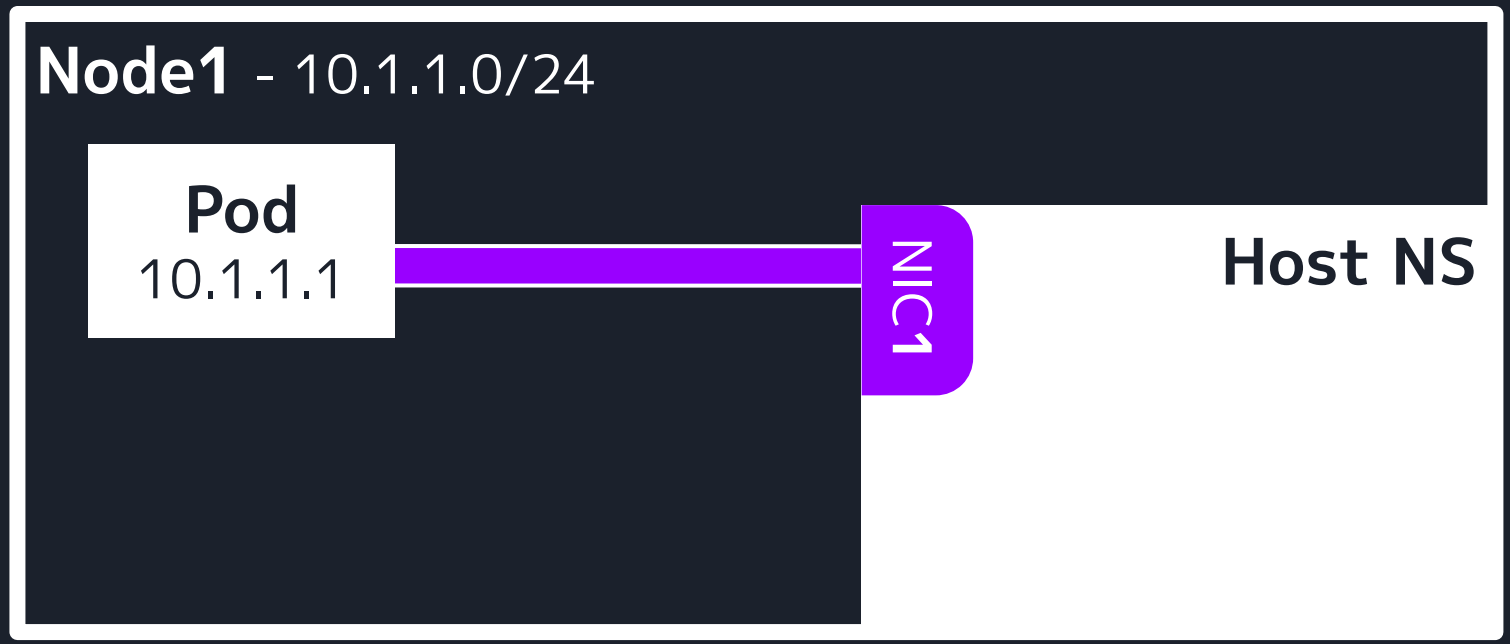
下準備: Flannelと同じくノードに**サブネットを割り当て**

Calico - 同じノード内

Pod作成時

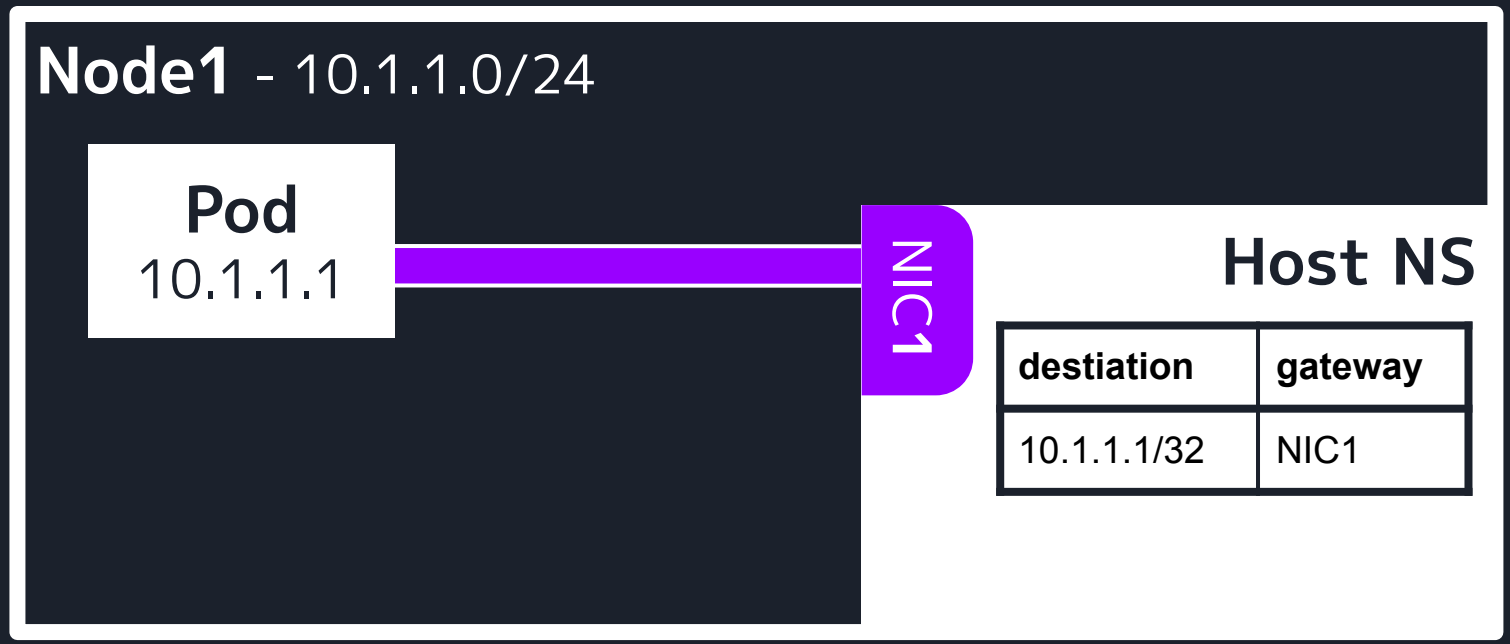
Calico - 同じノード内

Pod作成時: **veth**でHost Namespaceと**直接接続**



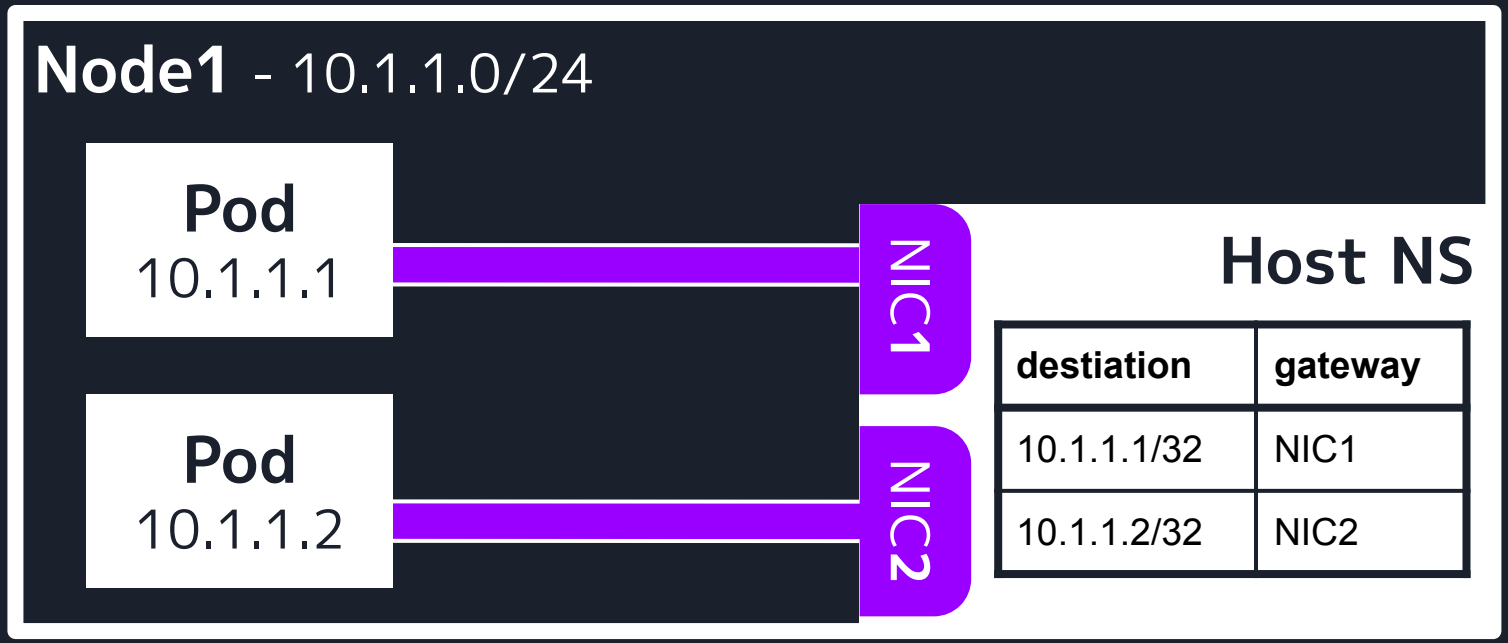
Calico - 同じノード内

Pod作成時: **ルーティングテーブル**にエントリを追加



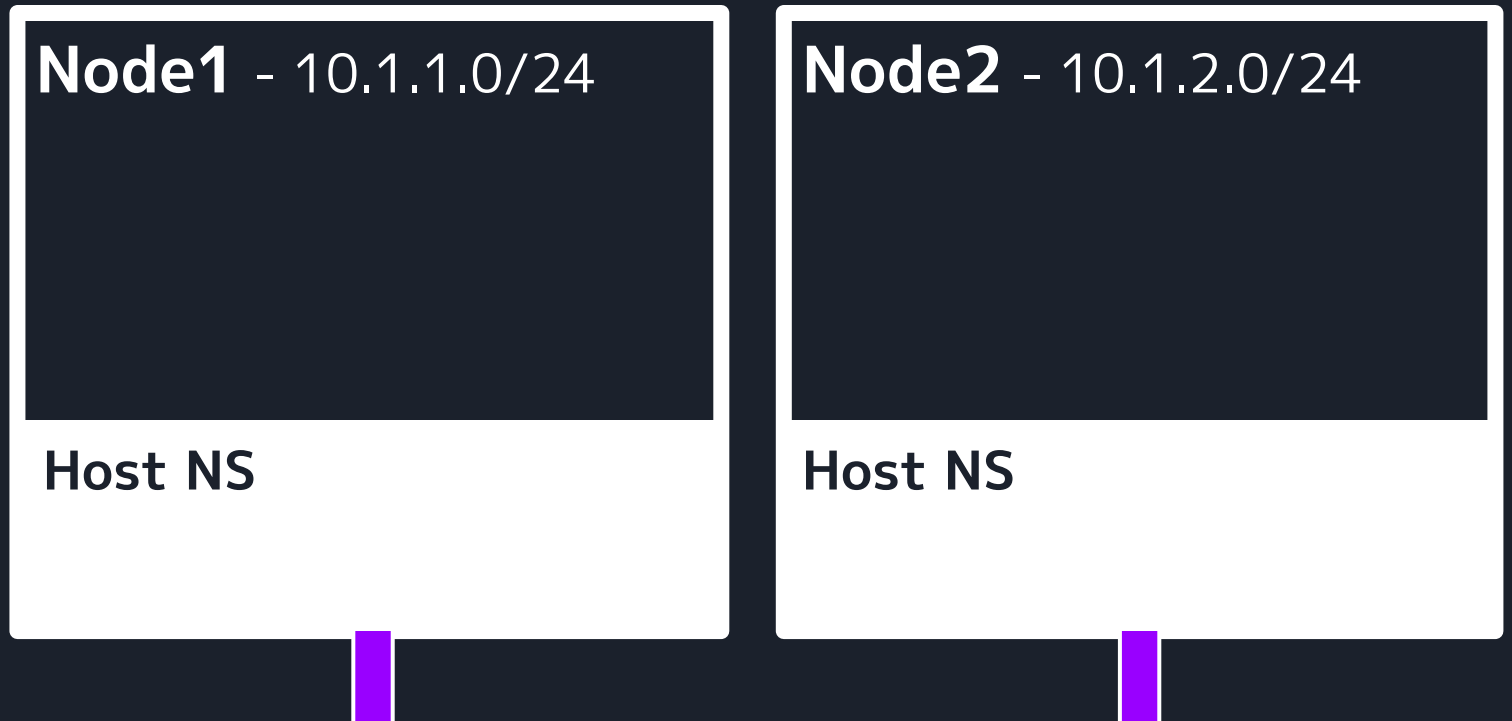
Calico - 同じノード内

通信時: **スタティックルーティング**でL3疎通ができる



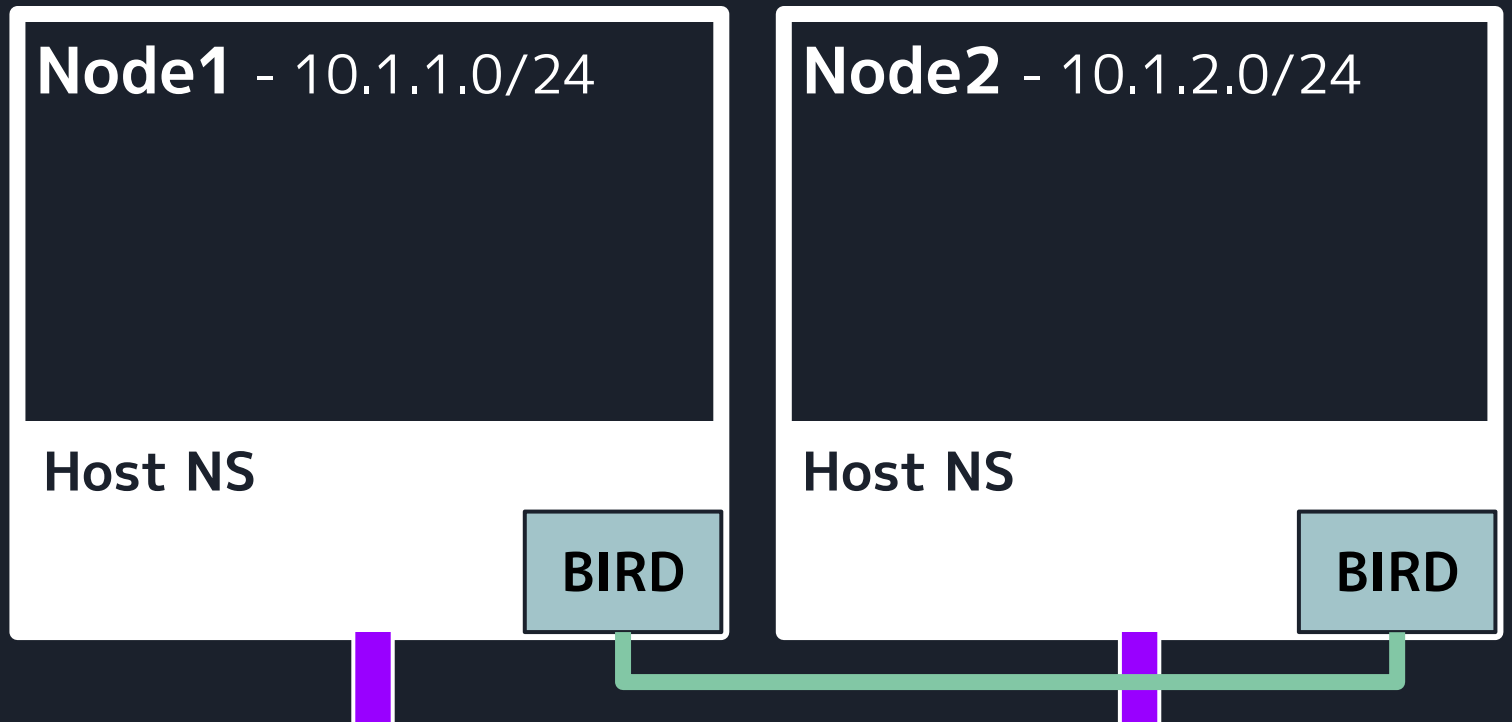
Calico - 異なるノード間

下準備: 同じノード内の下準備が終わった段階



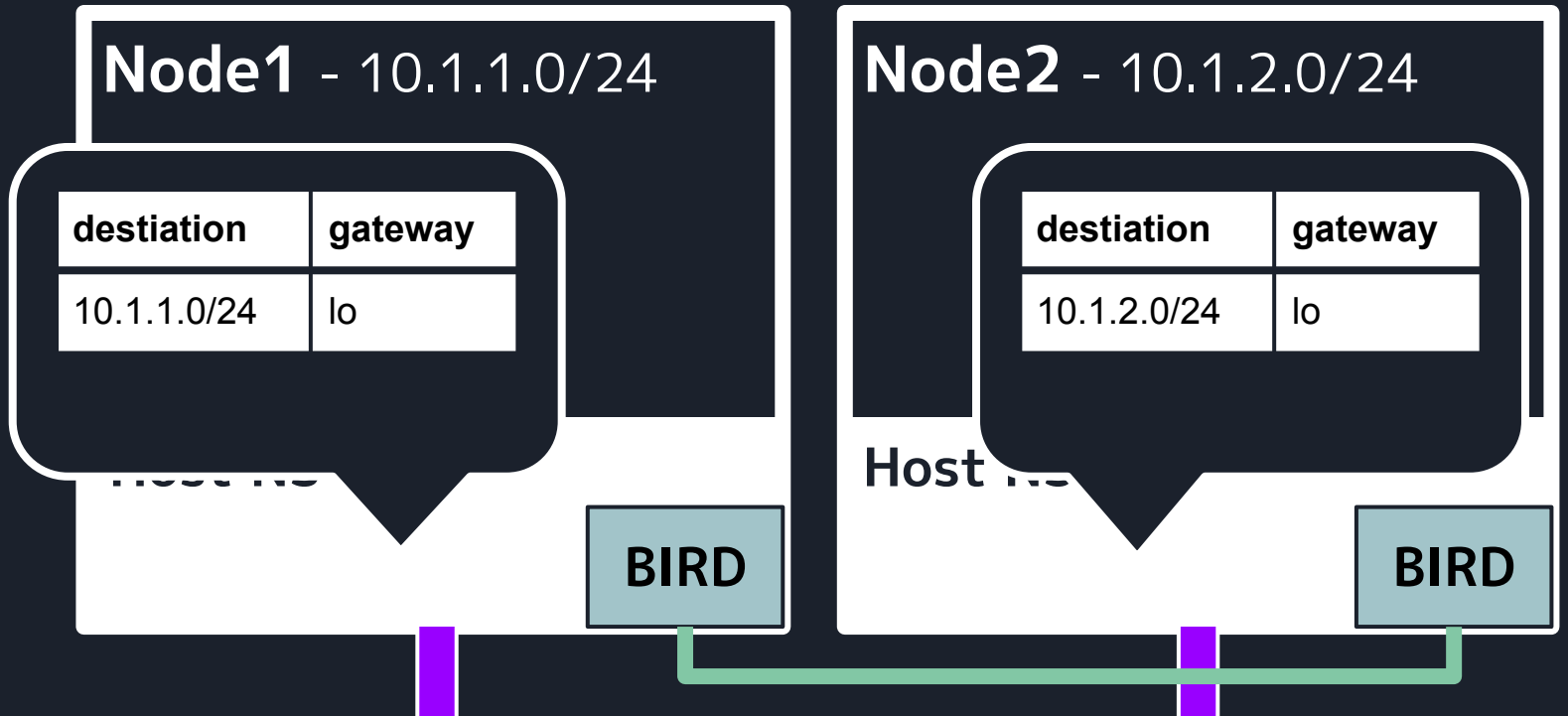
Calico - 異なるノード間

下準備: BIRD (ルーターソフトウェア) で**BGPピア**を張る



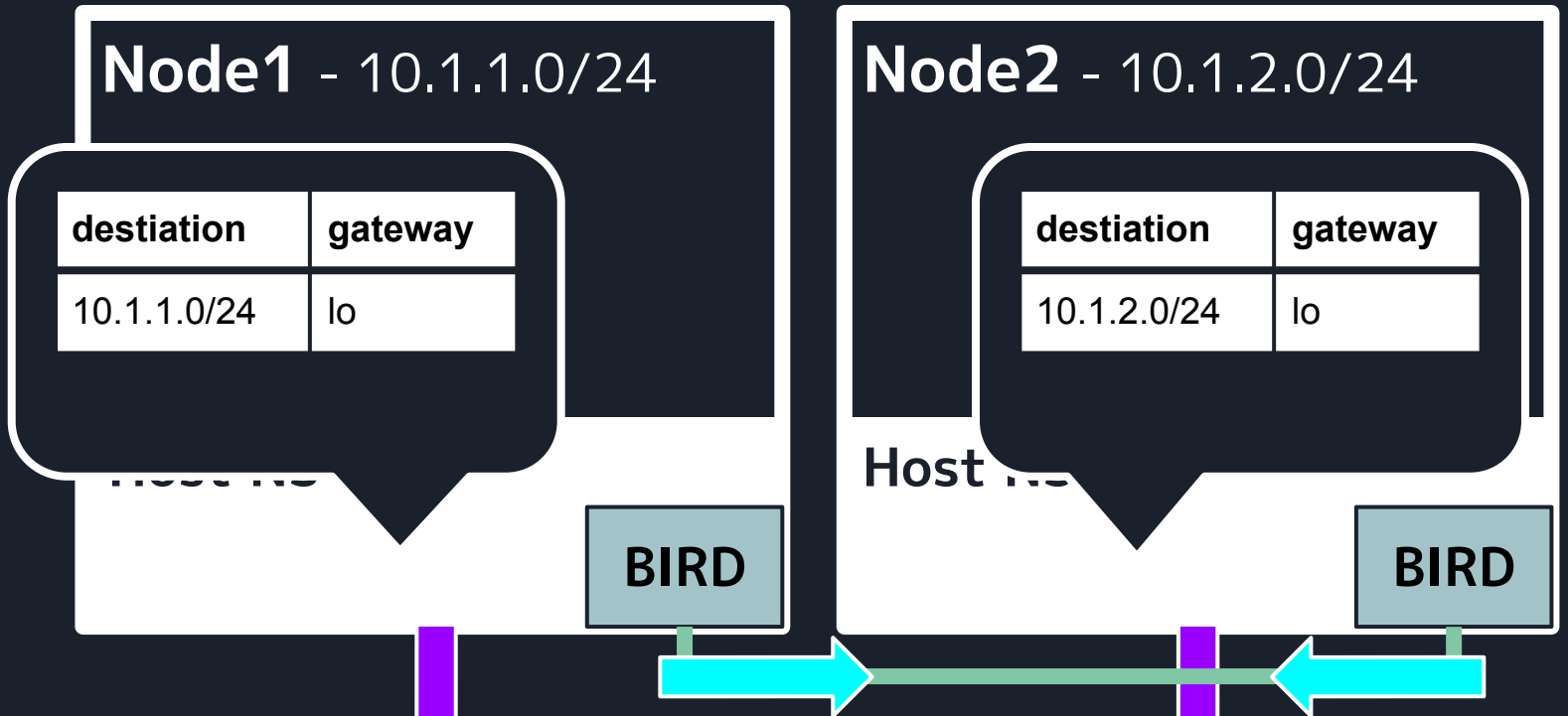
Calico - 異なるノード間

下準備: 自分のノードの**Podサブネット**を互いに**広報**する



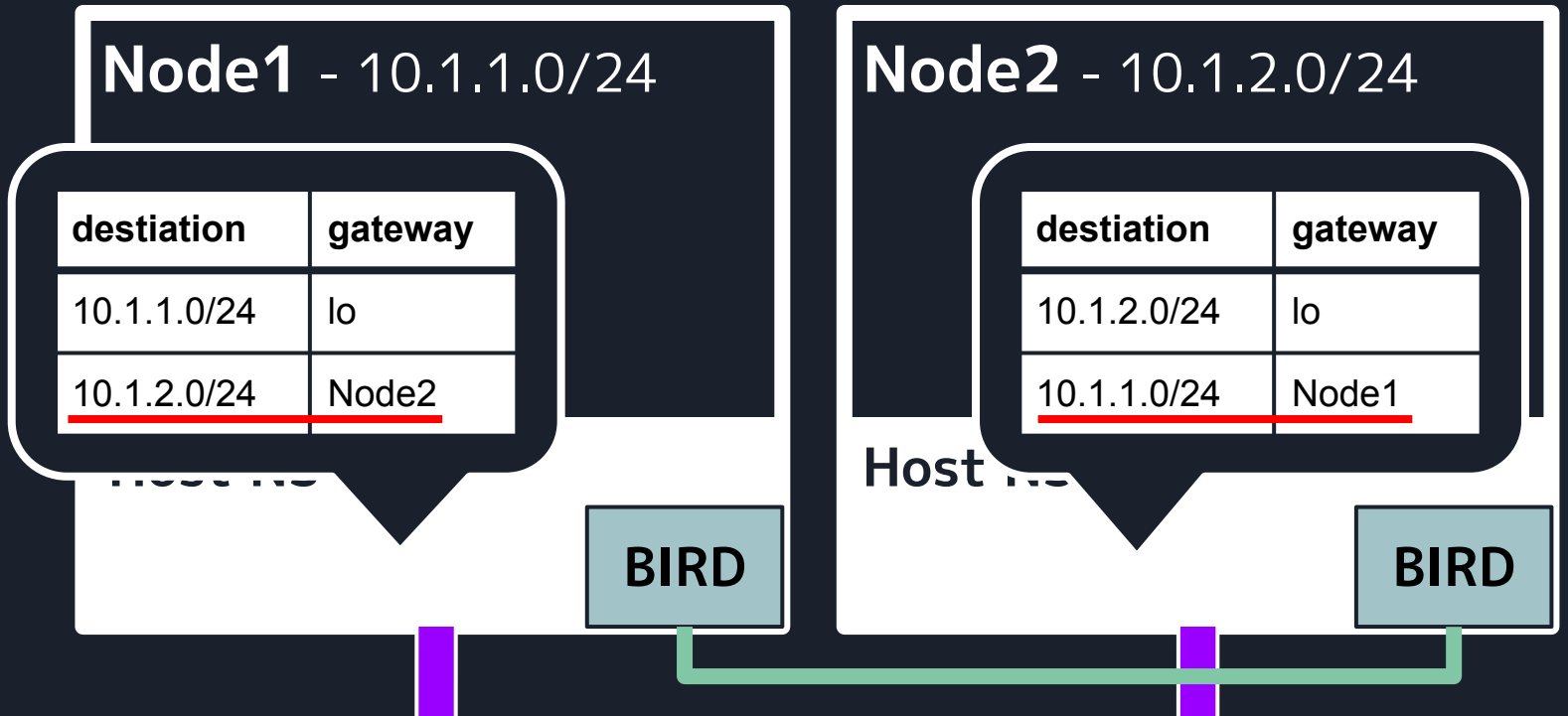
Calico - 異なるノード間

下準備: 自分のノードの**Podサブネット**を互いに**広報**する



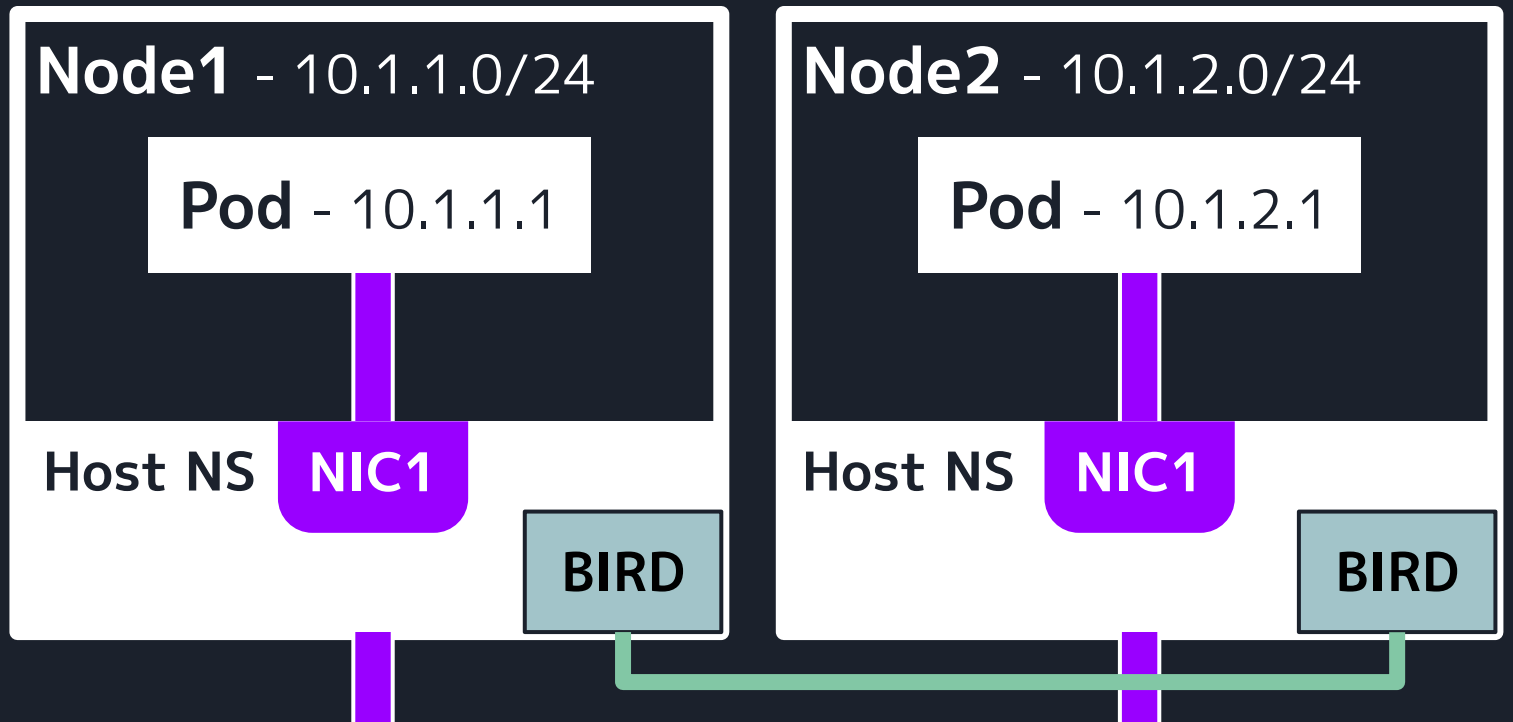
Calico - 異なるノード間

下準備: 自分のノードの**Podサブネット**を互いに**広報**する



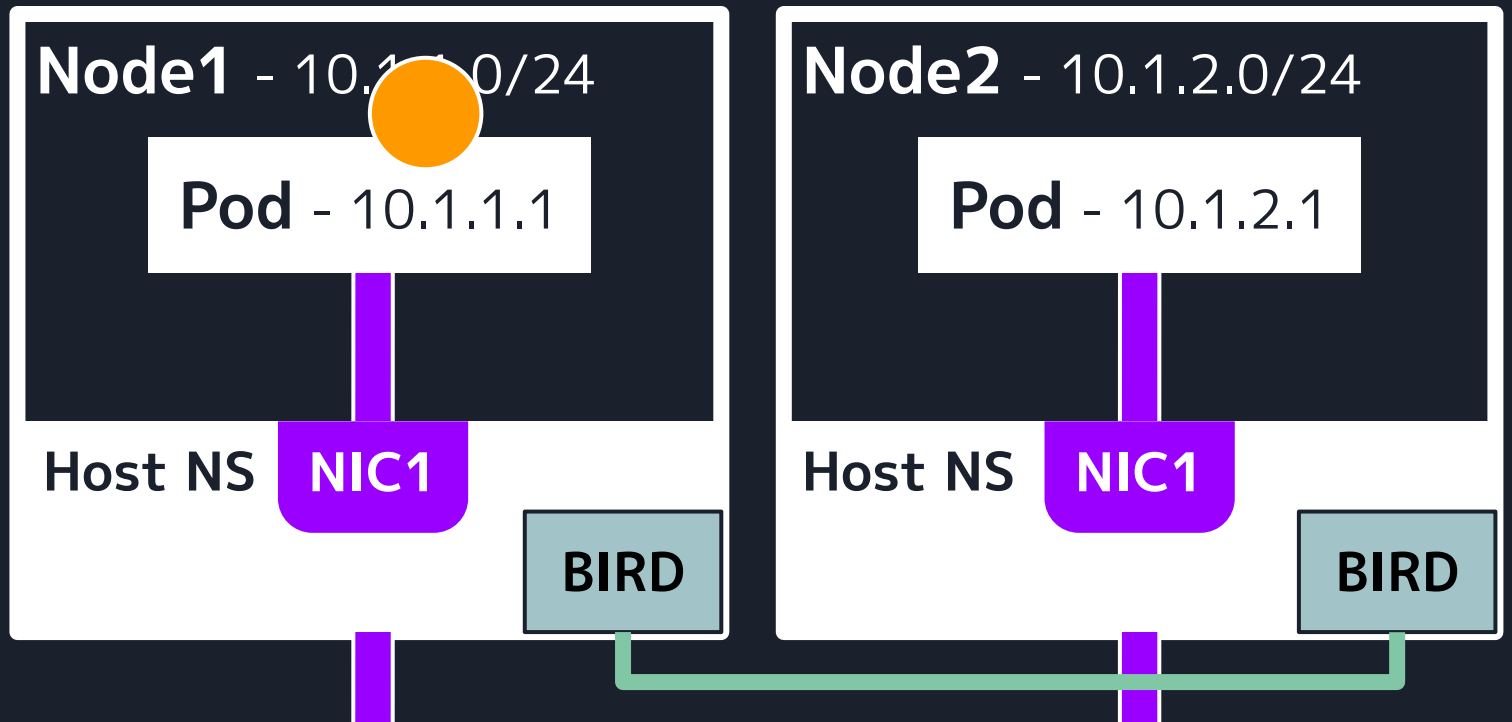
Calico - 異なるノード間

Pod作成時: 追加操作は特になし



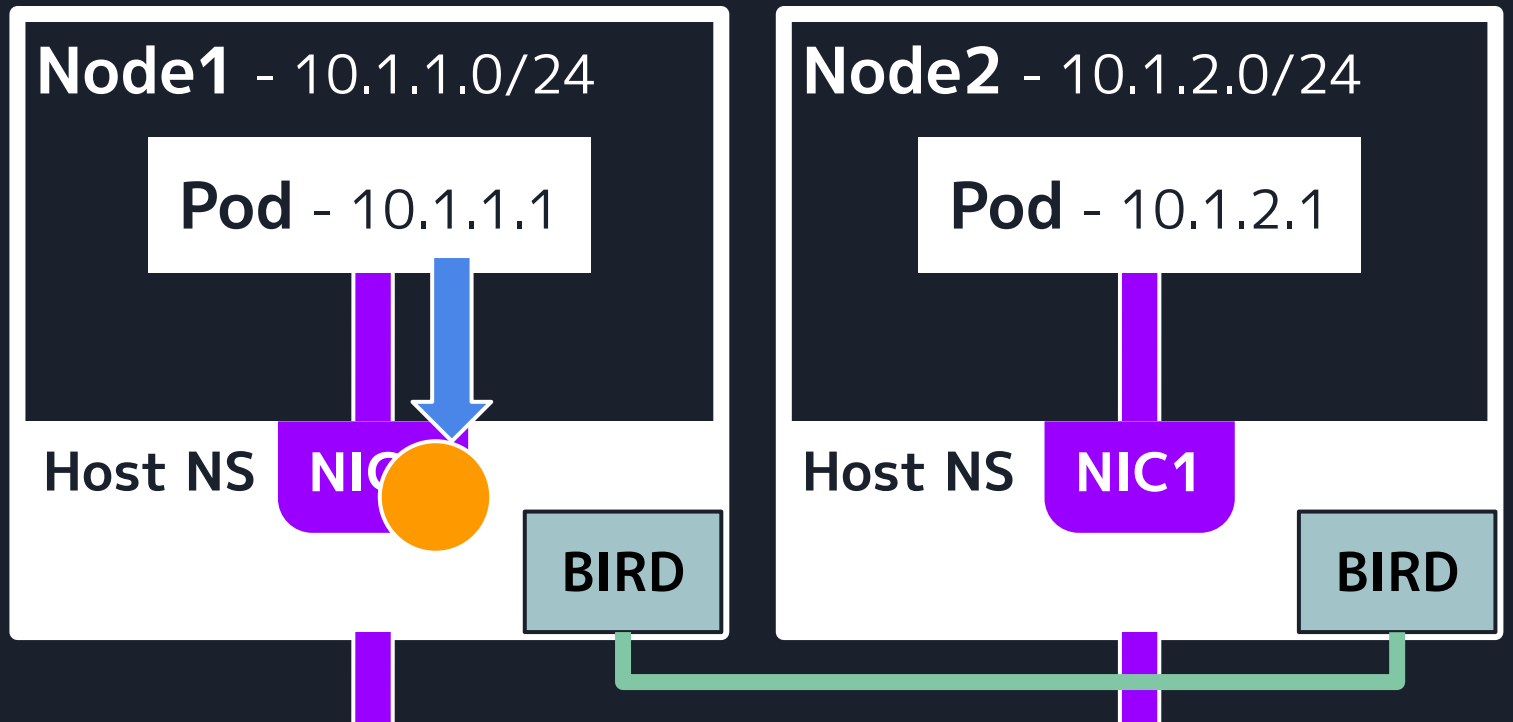
Calico - 異なるノード間

通信時: **10.1.1.1**から**10.1.2.1**へ送る時



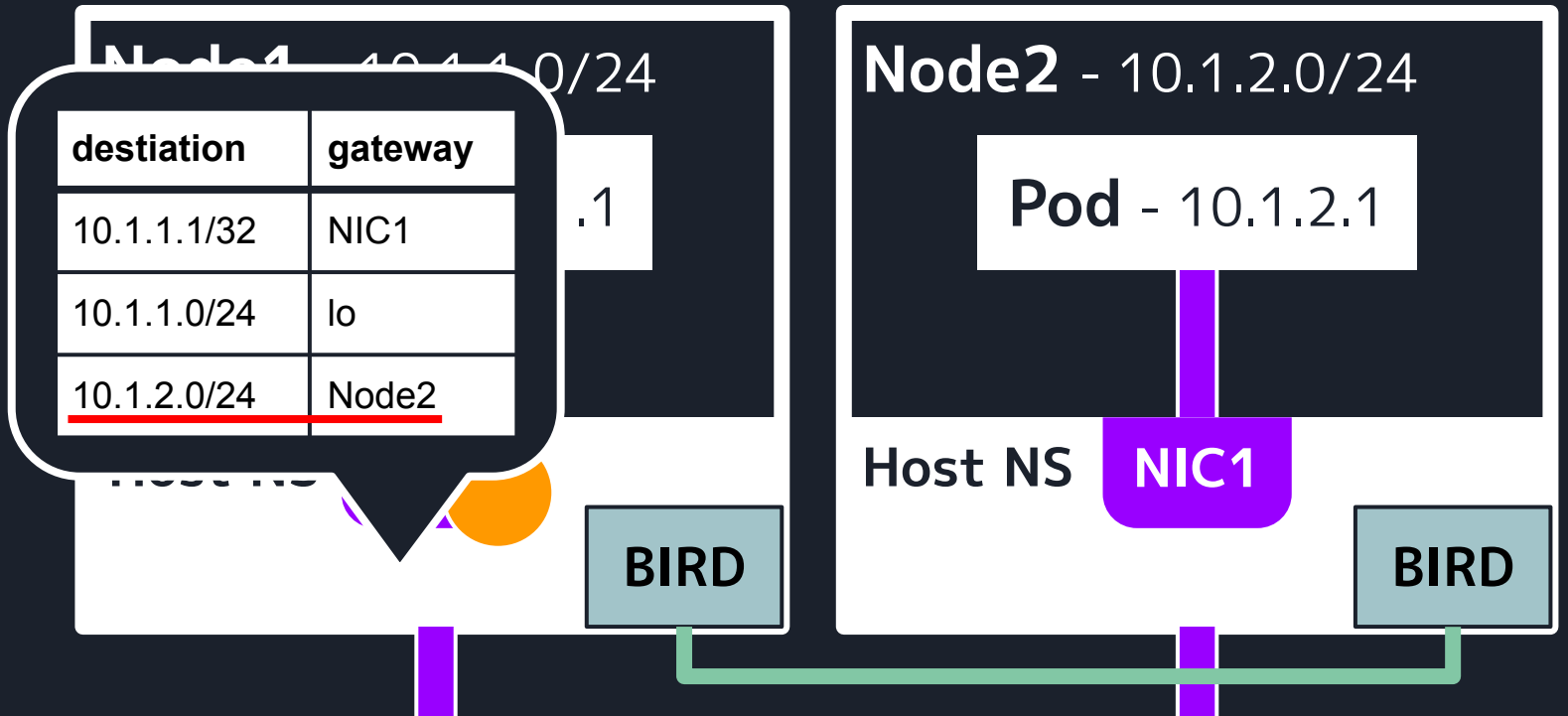
Calico - 異なるノード間

通信時: **デフォルトゲートウェイ**となっているHost NSへ



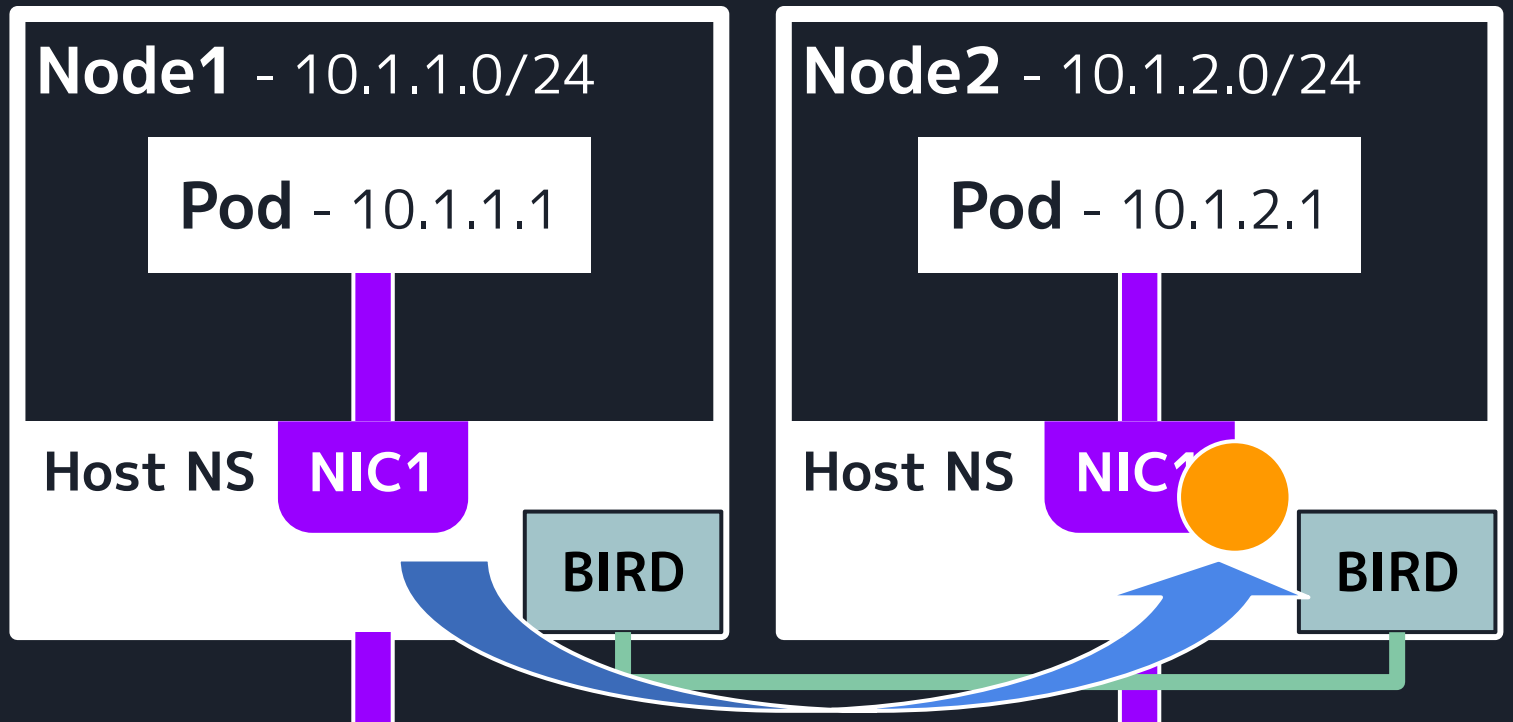
Calico - 異なるノード間

通信時: **ルーティングテーブル**を確認



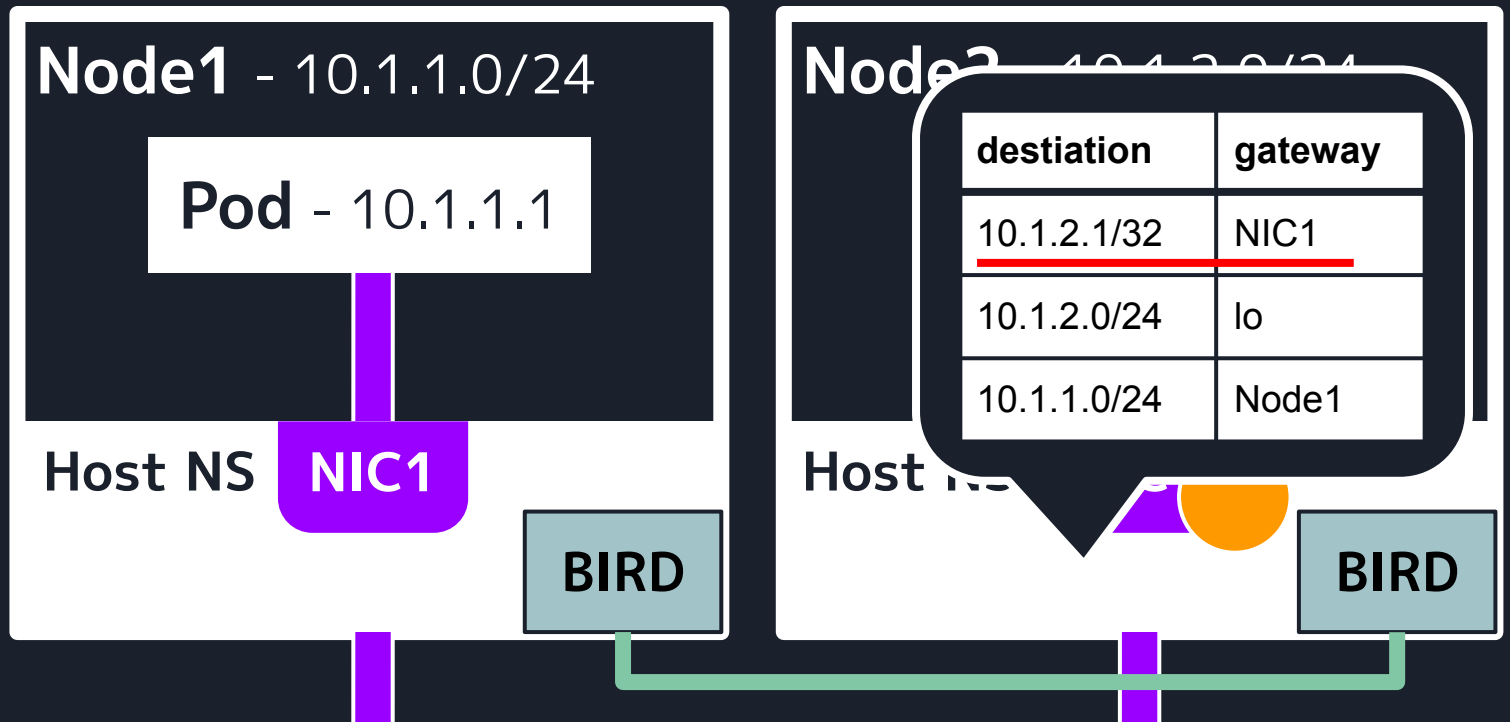
Calico - 異なるノード間

通信時: **アンダーレイから直接**、gatewayのNode2へ送信



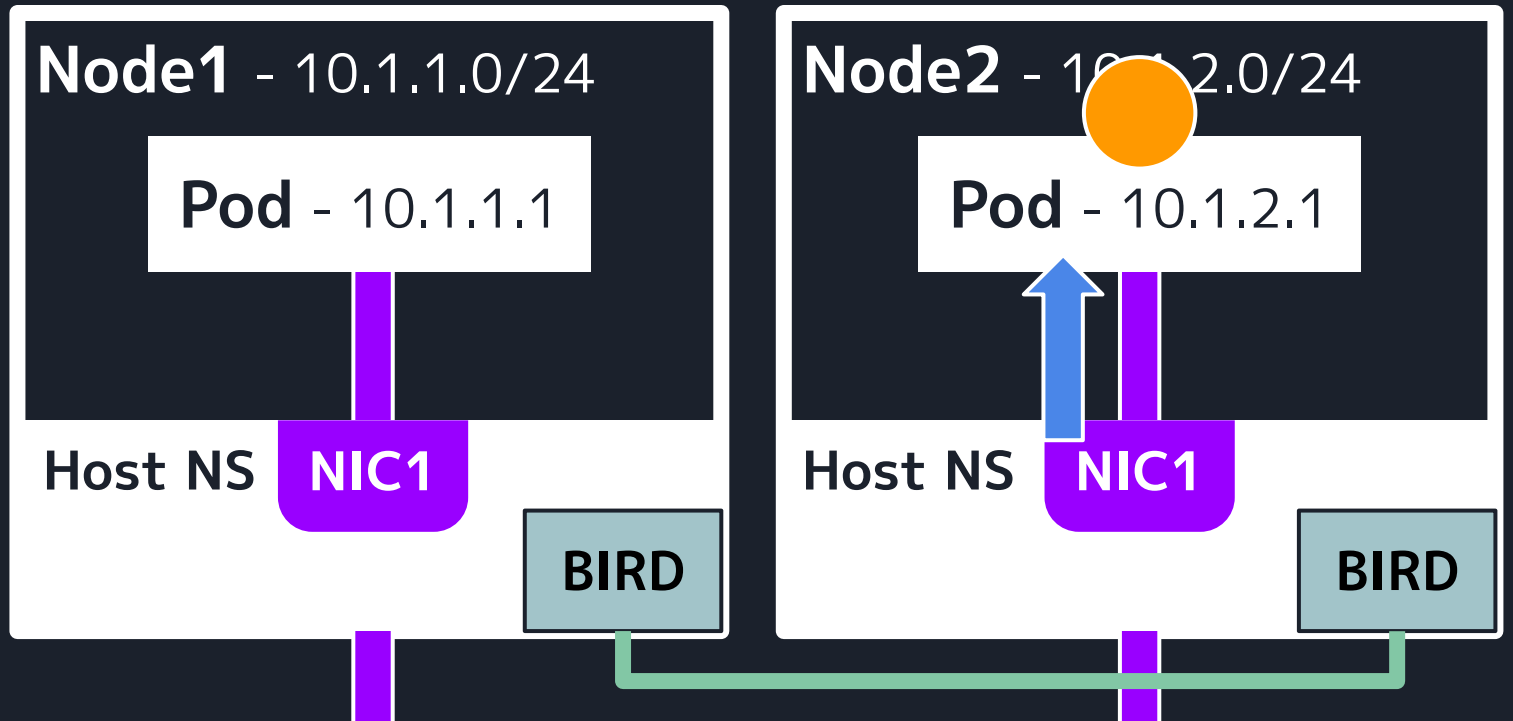
Calico - 異なるノード間

通信時: **ルーティングテーブル**を確認



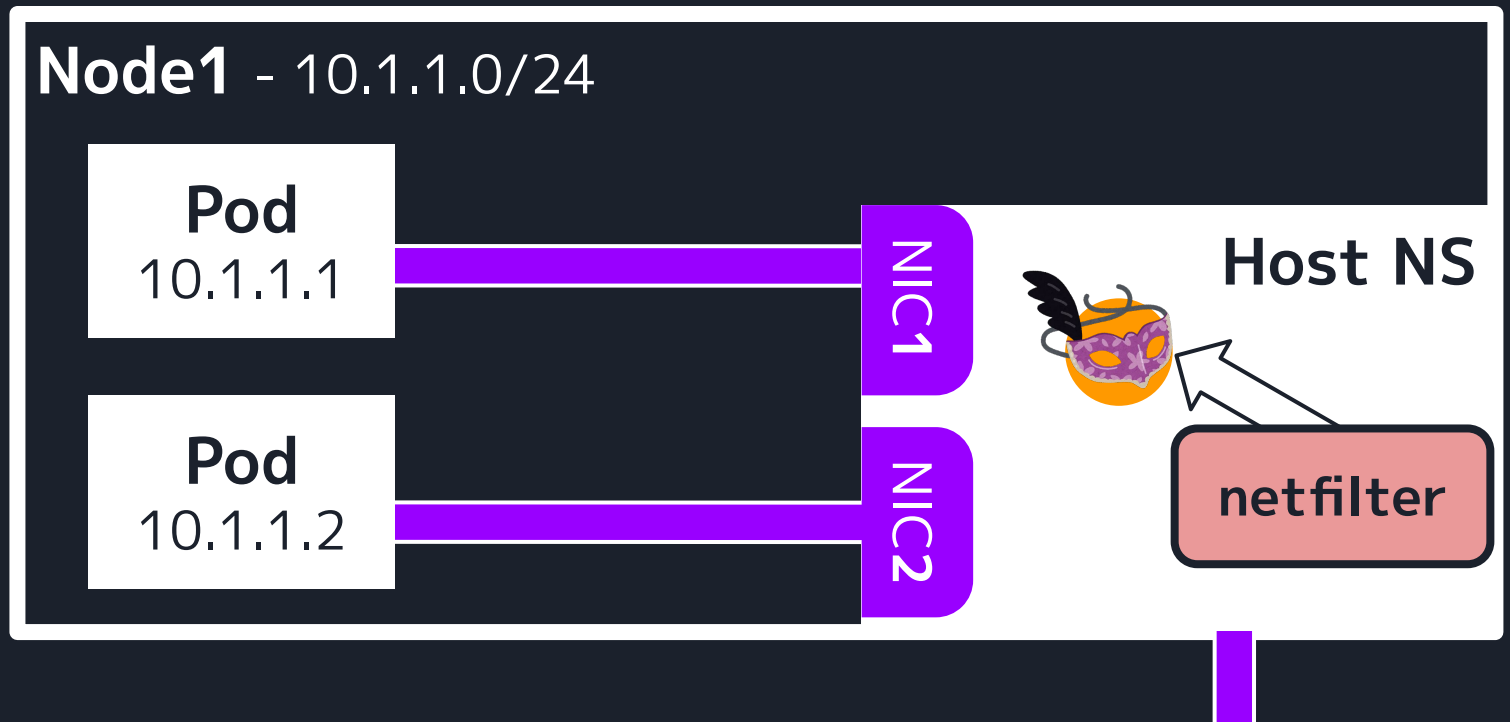
Calico - 異なるノード間

通信時: gatewayの**NIC1**から宛先に到達



Calico - Podから外部ネットワーク

Flannel同様、Netfilterで**IPマスカレード**をして外部に出す



Calico - 特徴まとめ

- **同じノード内**でのL3疎通
  - **スタティックルーティング**する
- **異なるノード間**のL3疎通
  - **BGP**で経路を広報、直接相手ノードに送る**Pure L3**
    - オーバーレイよりも**高効率**
  - 【制約】全てのノードが**同じL2セグメント**に属している必要がある
- それ以外はFlannel同様

Cilium - 同じノード内

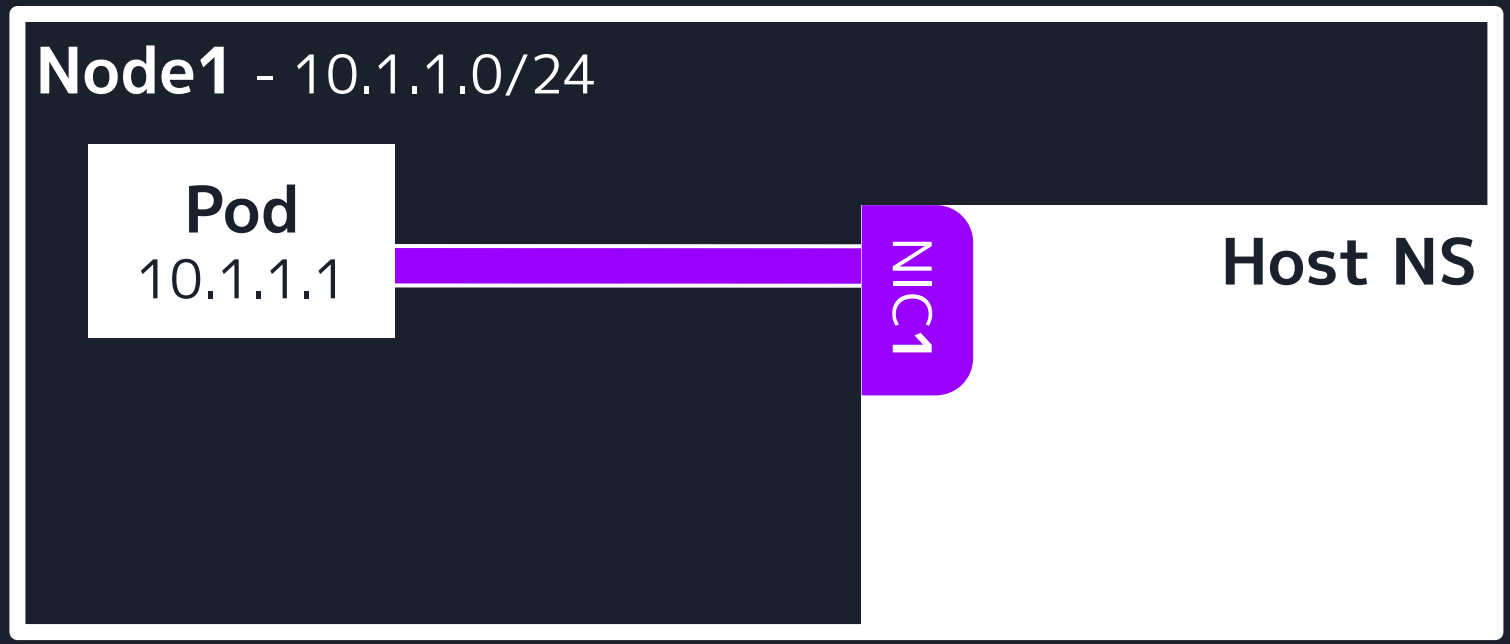
下準備: 前二つと同じくノードに**サブネットを割り当て**

Cilium - 同じノード内

Pod作成時

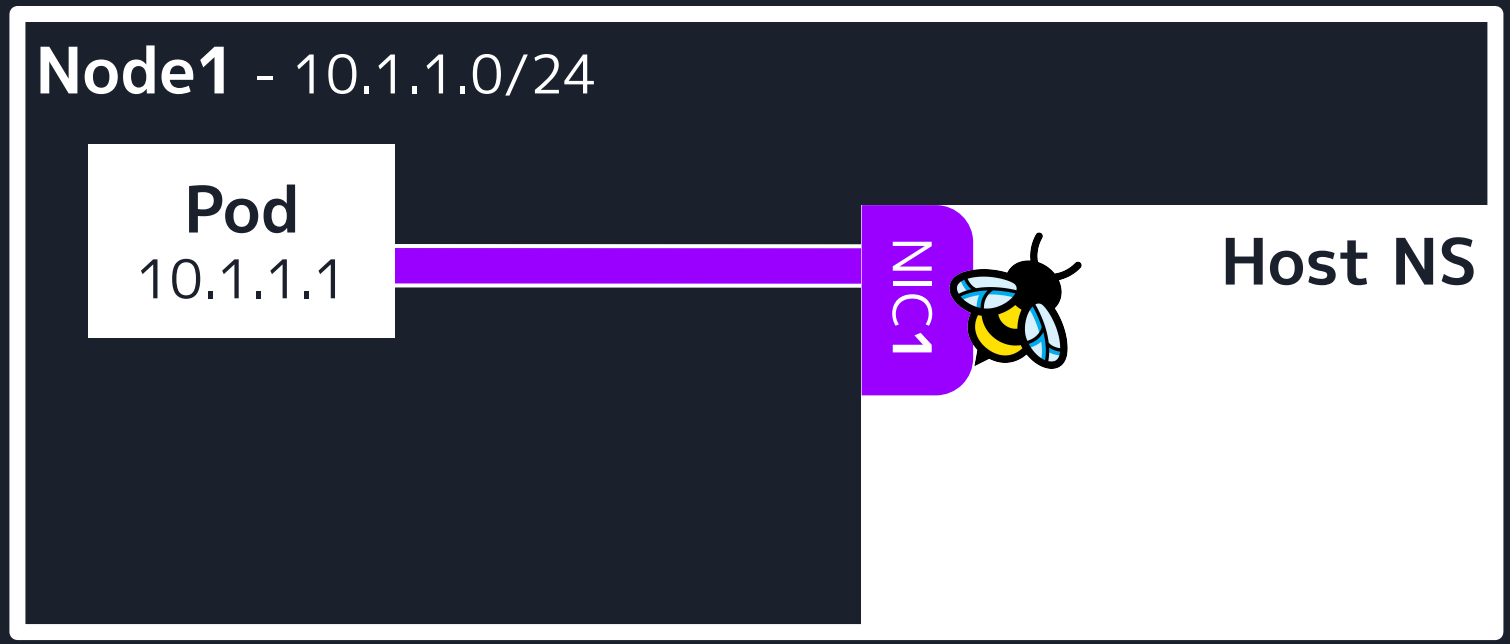
Cilium - 同じノード内

Pod作成時: **veth**でHost Namespaceと**直接接続**



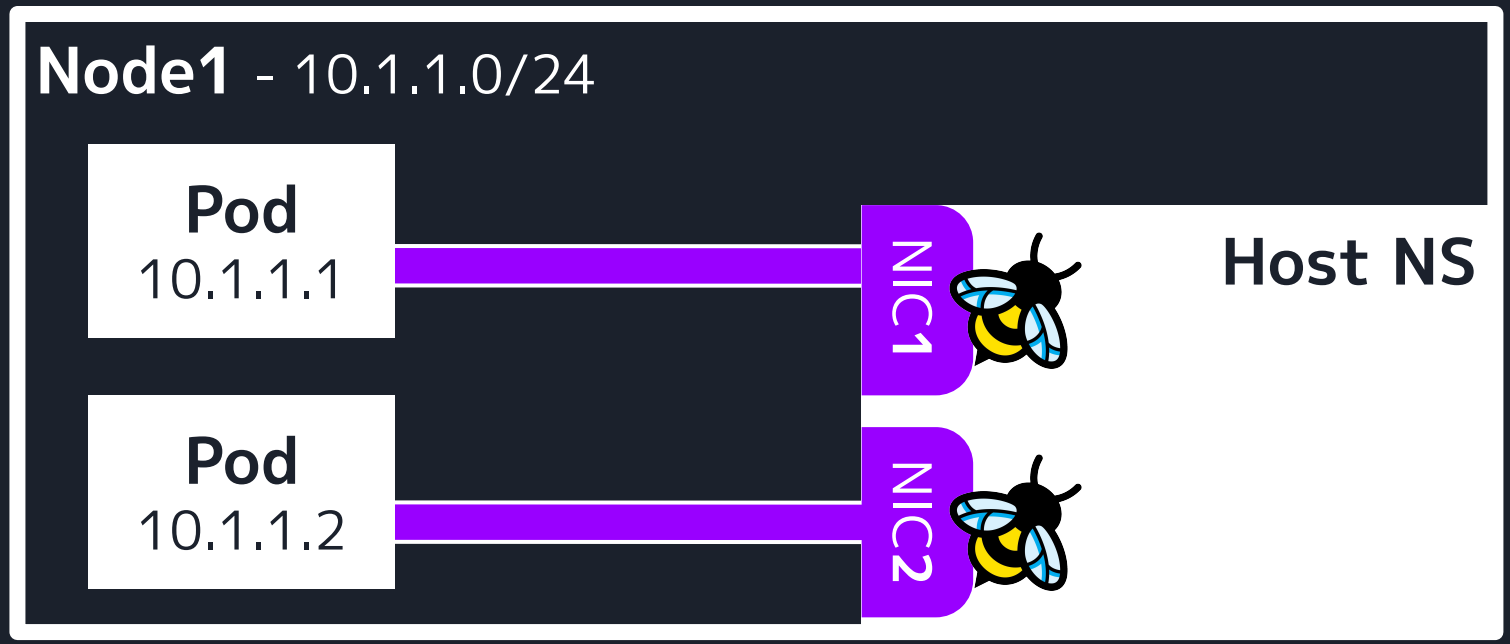
Cilium - 同じノード内

Pod作成時: Host側のNICに**eBPF**がアタッチされる



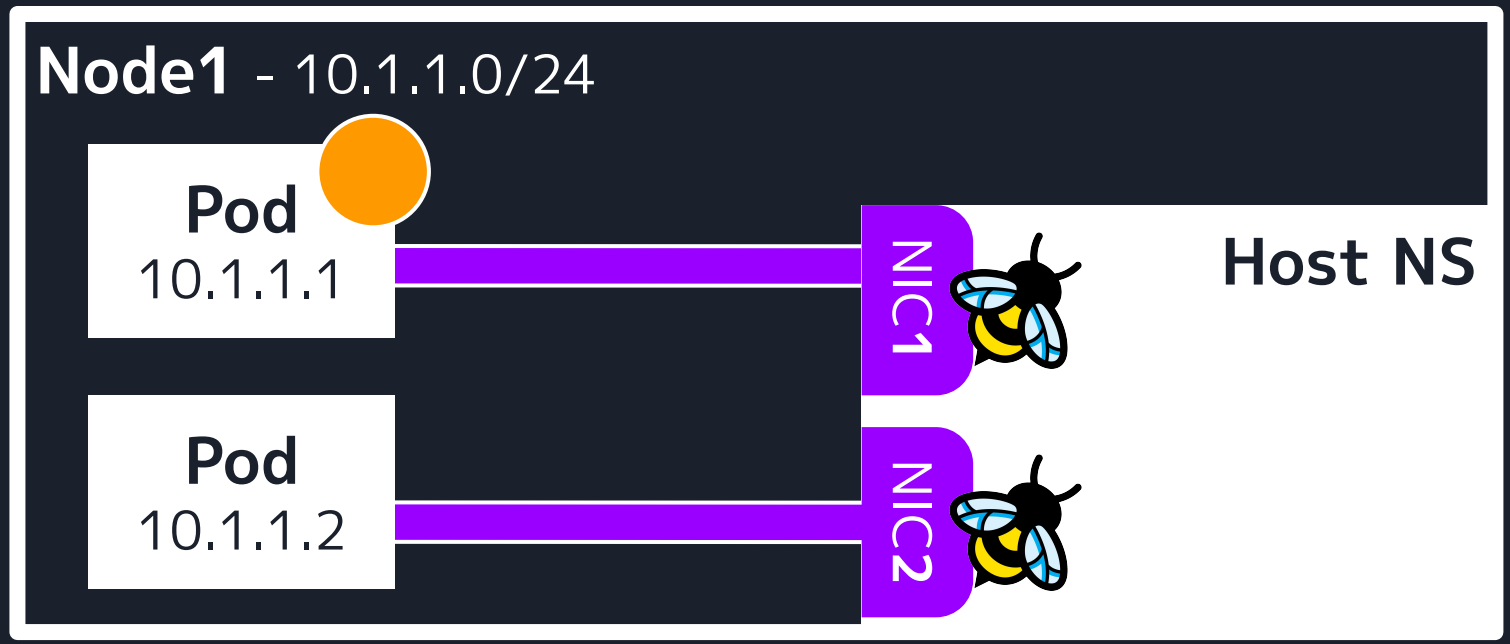
Cilium - 同じノード内

Pod作成時: 別Podも同様



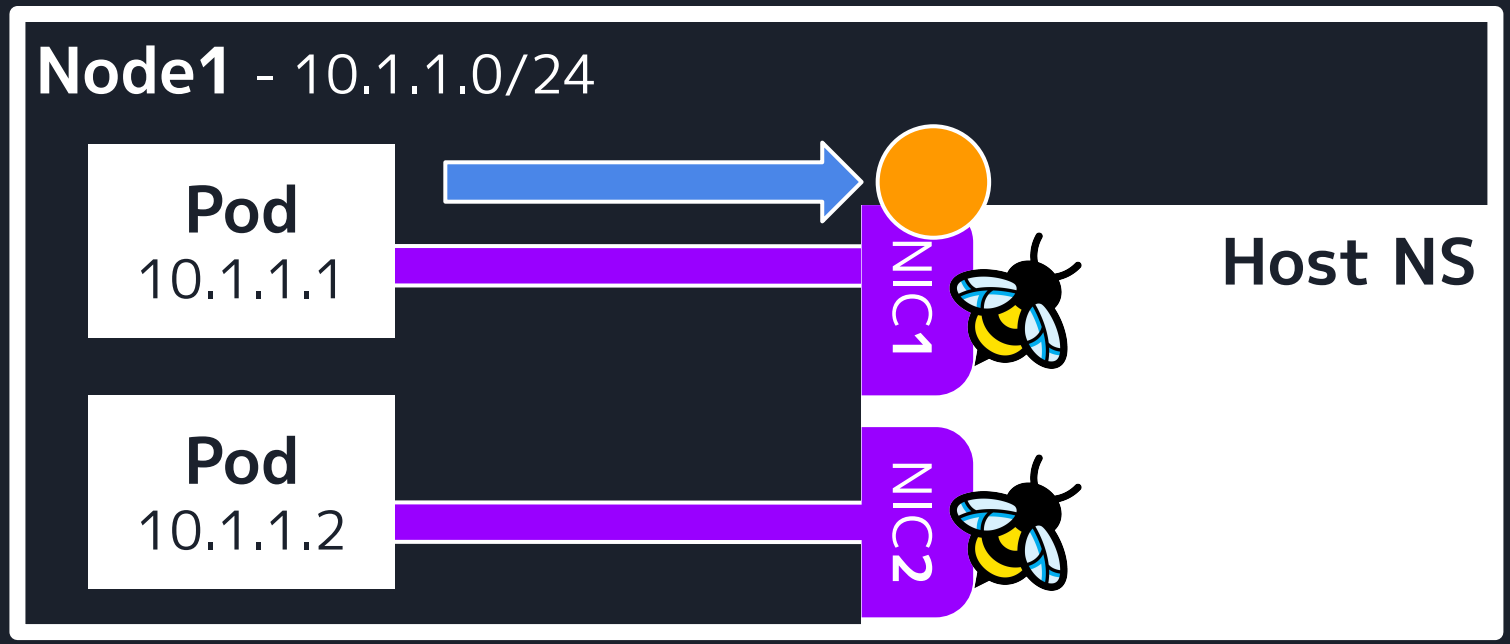
Cilium - 同じノード内

通信時: **10.1.1.1**から**10.1.1.2**へ送る時



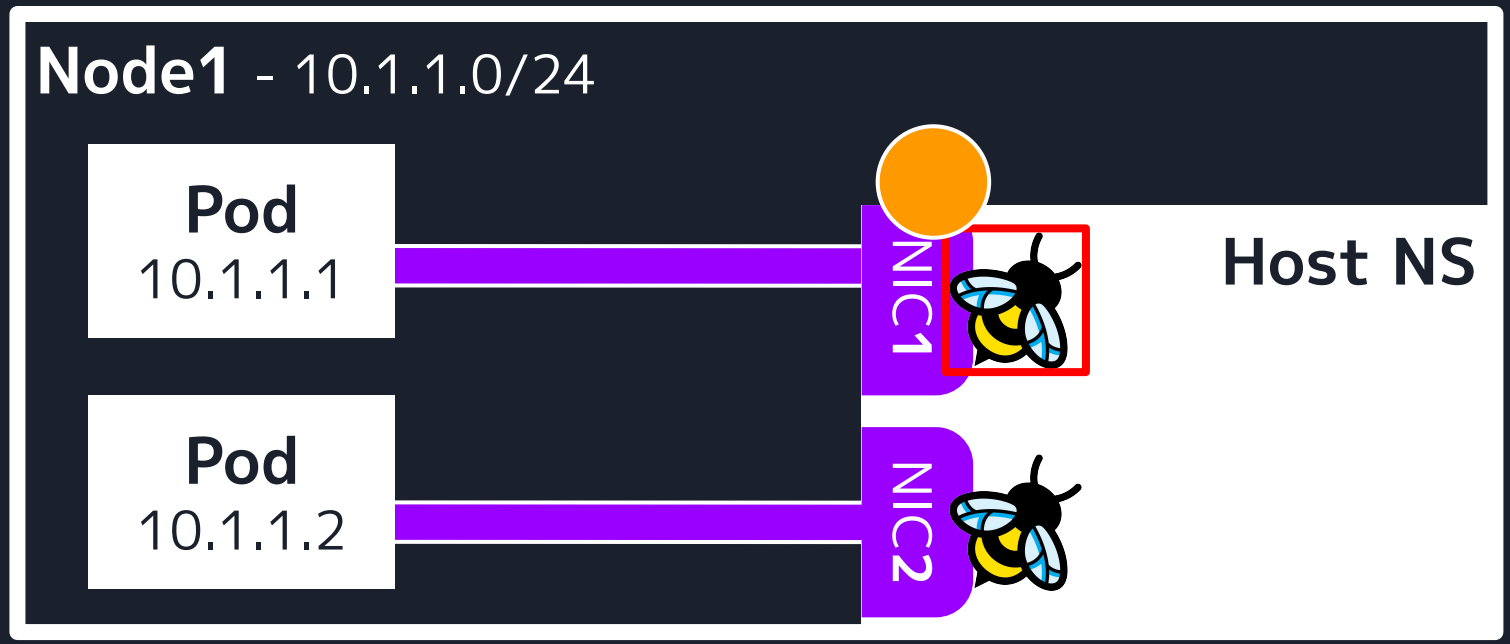
Cilium - 同じノード内

通信時: **デフォルトゲートウェイ**となっているHost NSへ



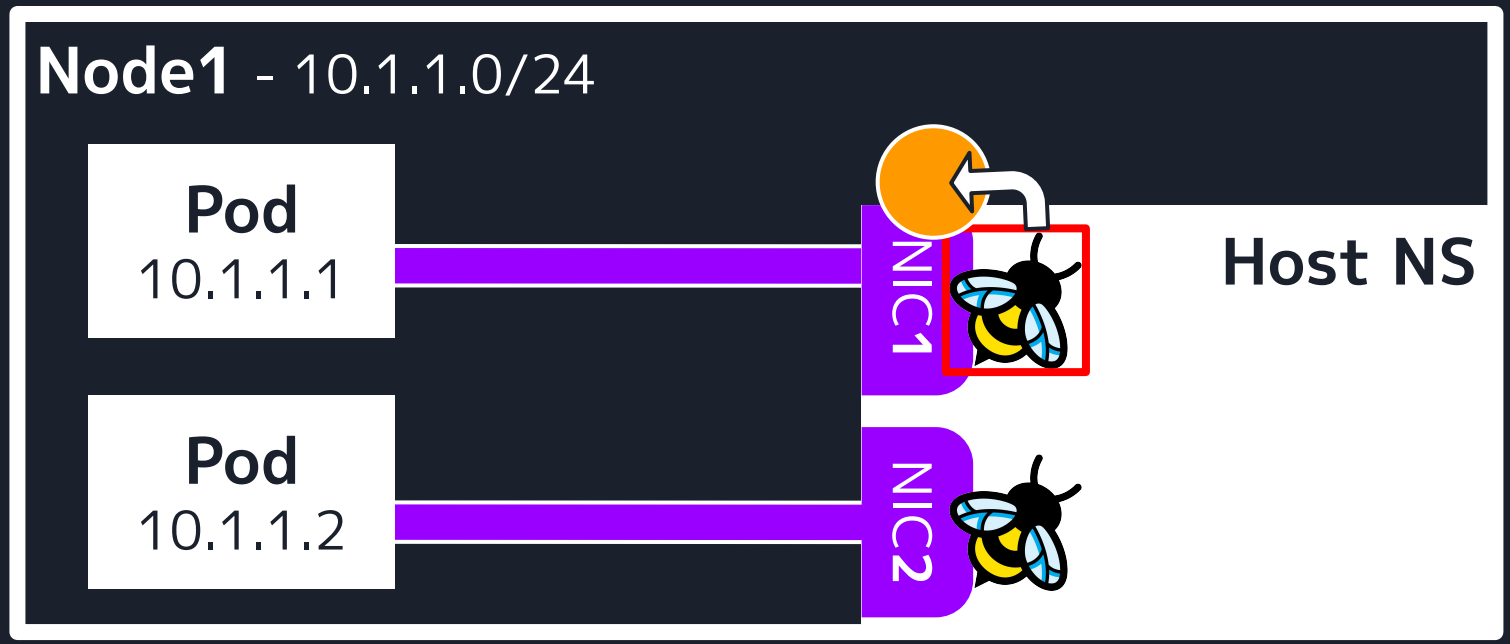
Cilium - 同じノード内

通信時: パケットがNICに着くと**eBPFプログラム**が起動



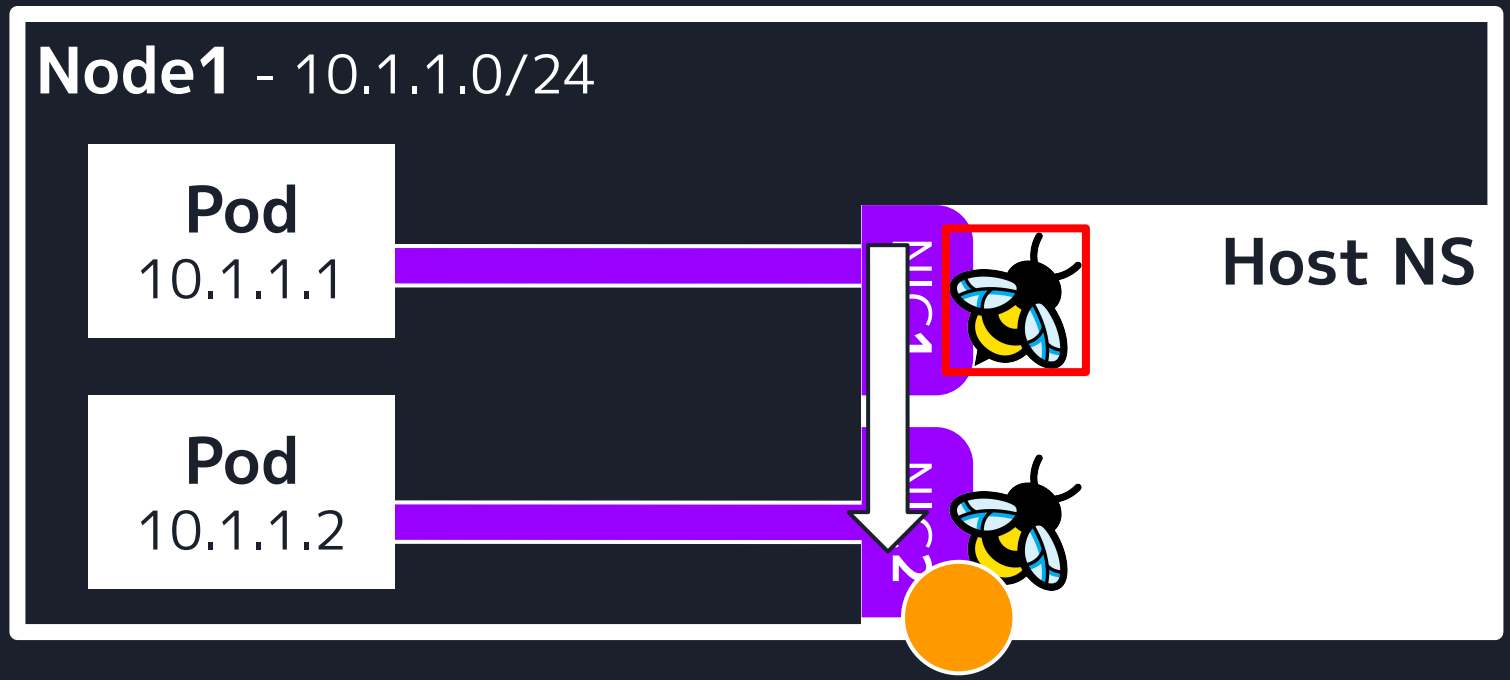
Cilium - 同じノード内

通信時: 宛先MACを**宛先Pod**のMACに書き換える



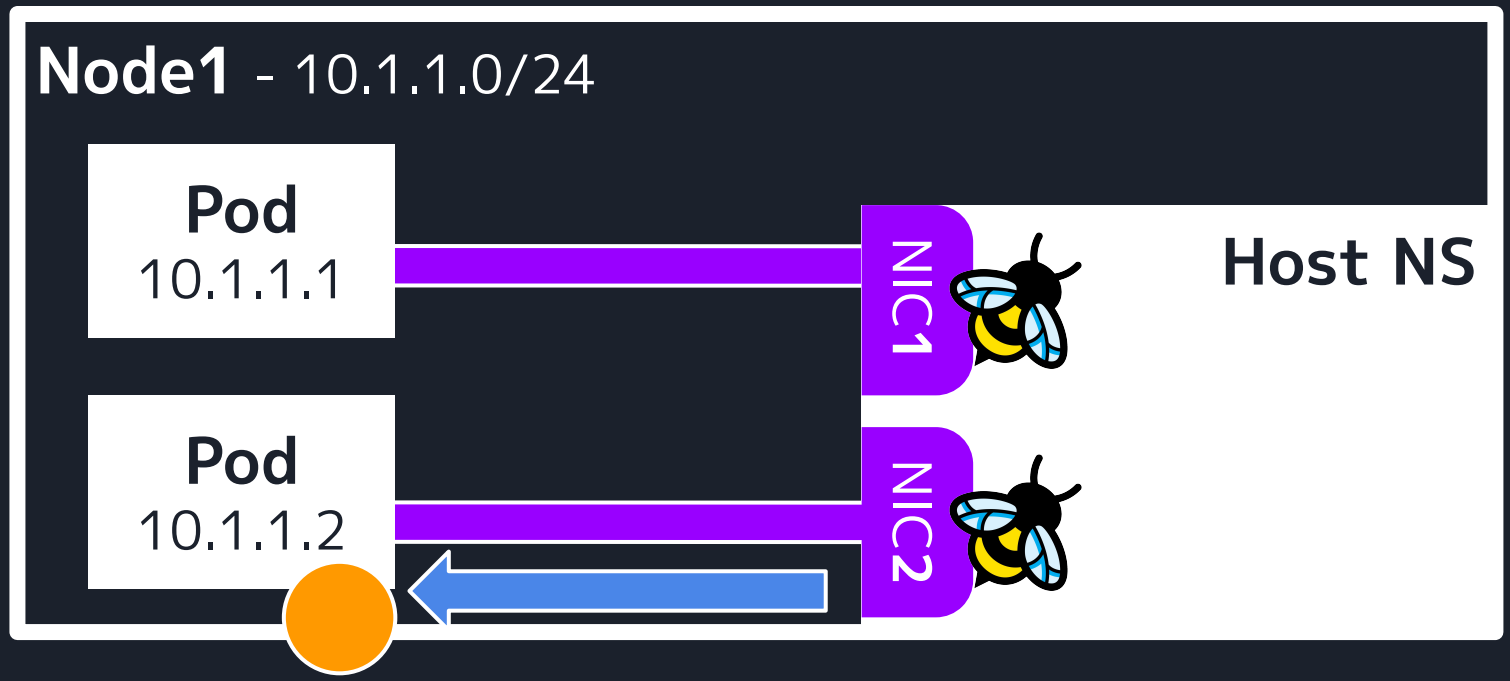
Cilium - 同じノード内

通信時: eBPFの機能で**NIC2に直接転送**



Cilium - 同じノード内

通信時: あとはそのまま宛先Podへ

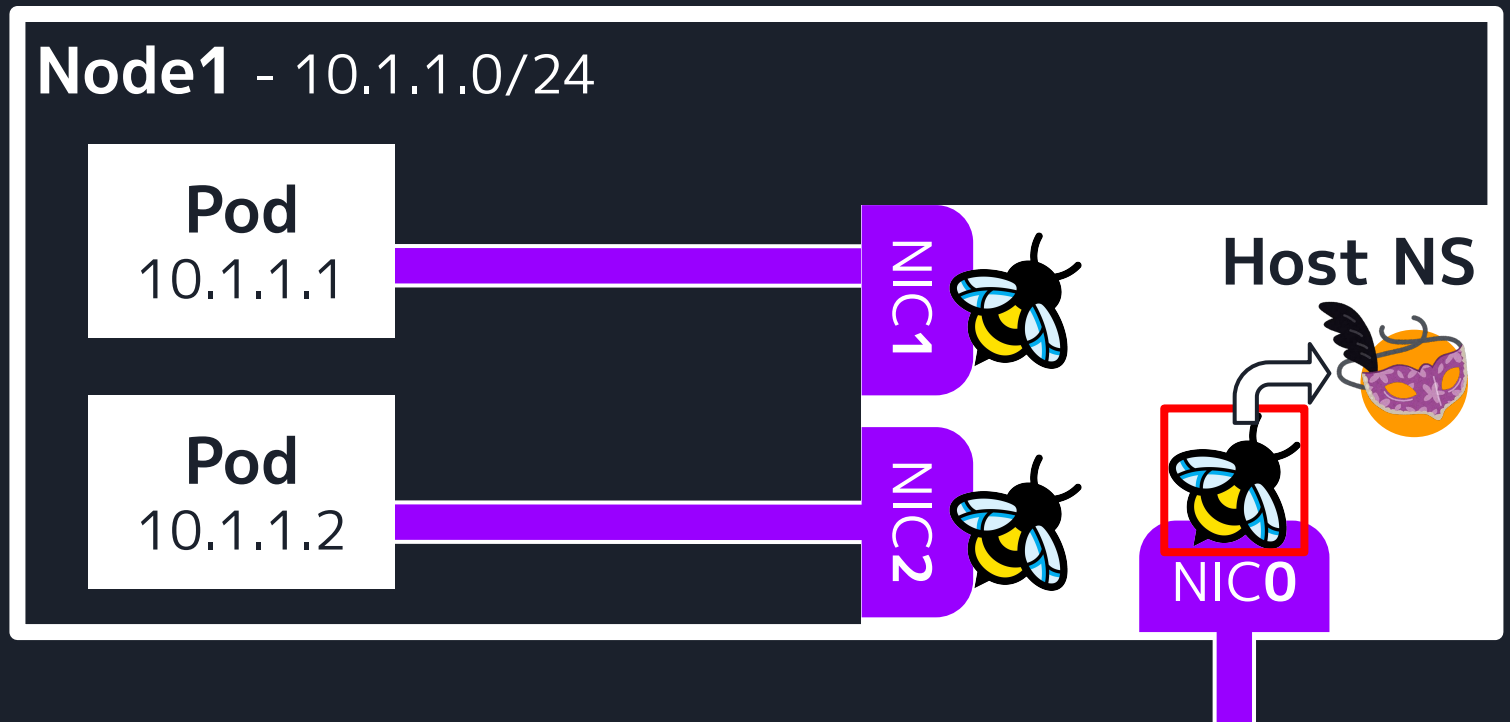


Cilium - 異なるノード間

- Ciliumは、同じノードのPod宛でなければ、eBPFプログラムを抜けて**Linux側に処理を投げる**
- Ciliumはあらかじめ、異なるノード間のパケット転送のために以下のいずれかを用意する
  - **Overlay**モード: Flannelと同じ**VXLAN**を使う転送
  - **Native**モード: Calicoと同じ**BGP広報**を使う転送
- すなわち、Ciliumの異なるノード間疎通は**FlannelかCalicoと全く同じ仕組み**で行われる

Cilium - Podから外部ネットワーク

eBPFで**IPマスカレード**をして外部に出す



Cilium - 特徴まとめ

- **同じノード内**でのL3疎通
  - **eBPF**を使って高速にルーティングを行う
- **異なるノード間**のL3疎通
  - **Flannel / Calicoと同じ**仕組みを使う
  - 外からノードに入ってきたパケットは「同じノード内での疎通」と同じく**eBPF**でPodまで運ばれる
- **eBPF**でIPマスカレードする

ところで: 汎用的 ≠ それだけでいい

- これらのOSSは**汎用**を目指している
  - どんな環境でも動くように、**一般的な仕組み**のみを利用して実装をしている
- メジャーなOSSプラグインは、どんな所でも使える**汎用性**の代わりに**オーバーヘッド**を許容している
  - 特に**オーバーレイ**は顕著に遅い
  - **SDN**はそれなりにノードに処理を要求する

環境によって効率の良い形は違う

- なぜCNIはKubernetesにデフォルトで組み込まれずわざわざ**インターフェース**になっているのか
  - クラウドベンダーが自社のネットワークに合わせて**効率の良い処理基盤**を組むため
- CNIの魅力は、実装が完全に**ユーザー依存**であること
  - オンプレ環境でネットワークが全部わかるなら、それに**効率化**するに越したことはない

環境特化なCNIのインセンティブ

- **既存のアンダーレイ**ネットワークに**最適化**したい
  - 既にBGP/EVPN/VRF等、**成熟した基盤**があるとき
- 既存の**運用・監視・障害対応**フローに合わせたい
  - **既存の監視基盤**に乗せられる形でメトリクス・イベント・トレースを最初から設計したい
- Podをもっと**シームレスに**繋ぎたい
  - 既存のネットワークと深く統合すると**VMとPod**がシームレスに繋がったりする

環境特化なCNIプラグインの例

- **GKE Dataplane V2**
  - **GKEでデフォルト**になっているプラグイン
- **AWS VPC** CNI
  - EKSで使える、**VPCをベース**にしたプラグイン
- **Coil** on Neko
  - **Cybozu**が開発しているプラグイン

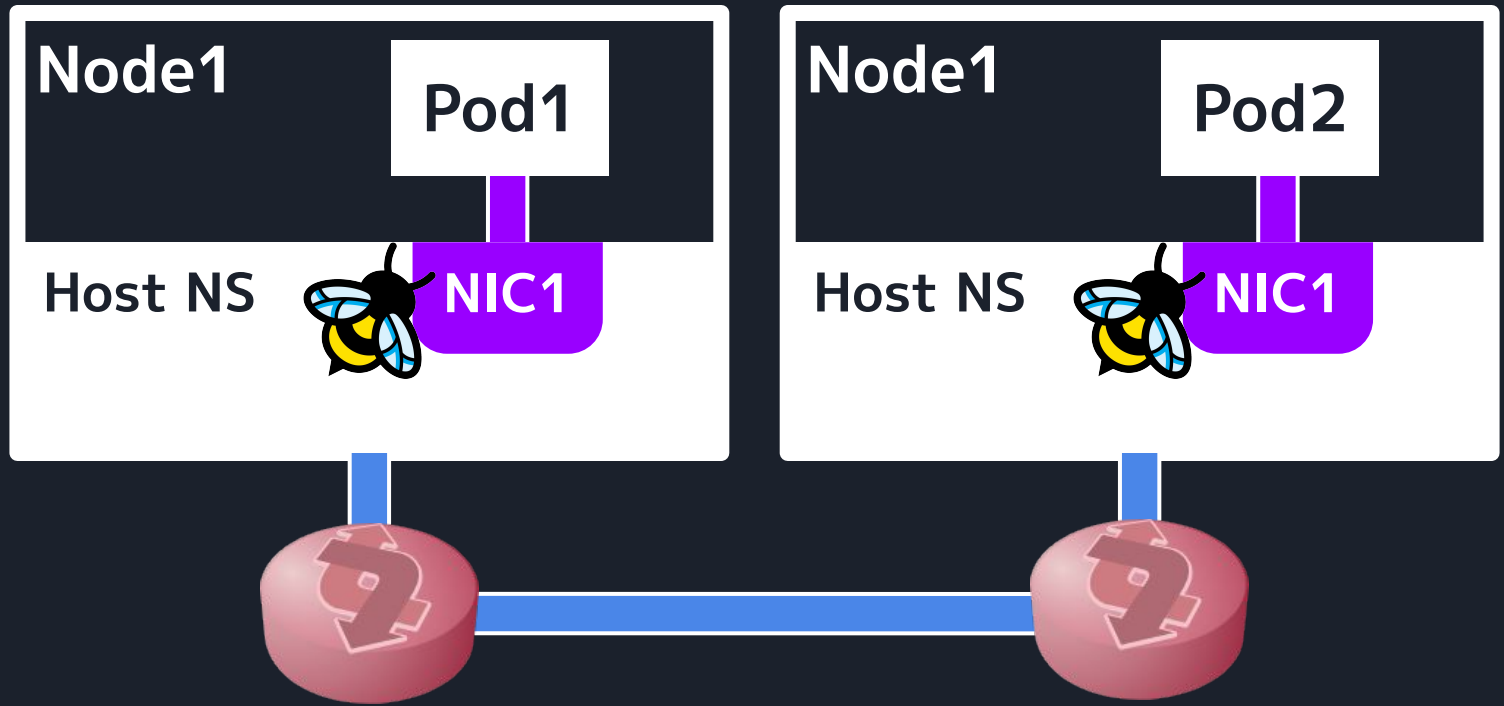
それぞれの概要を軽く見ていきましょう

GKE Dataplane V2

- GKEの開発チームが**Ciliumをカスタマイズ**したもの
- **同じノード内**でのL3疎通
  - **Cilium**の仕組みをそのまま用いる
- **異なるノード間**のL3疎通
  - 準備: **アンダーレイ**ネットワークに**設定を流し込む**
  - 同ノード内のPod宛でなければ**そのまま外に出す**
  - **アンダーレイ**ネットワークがルーティングする

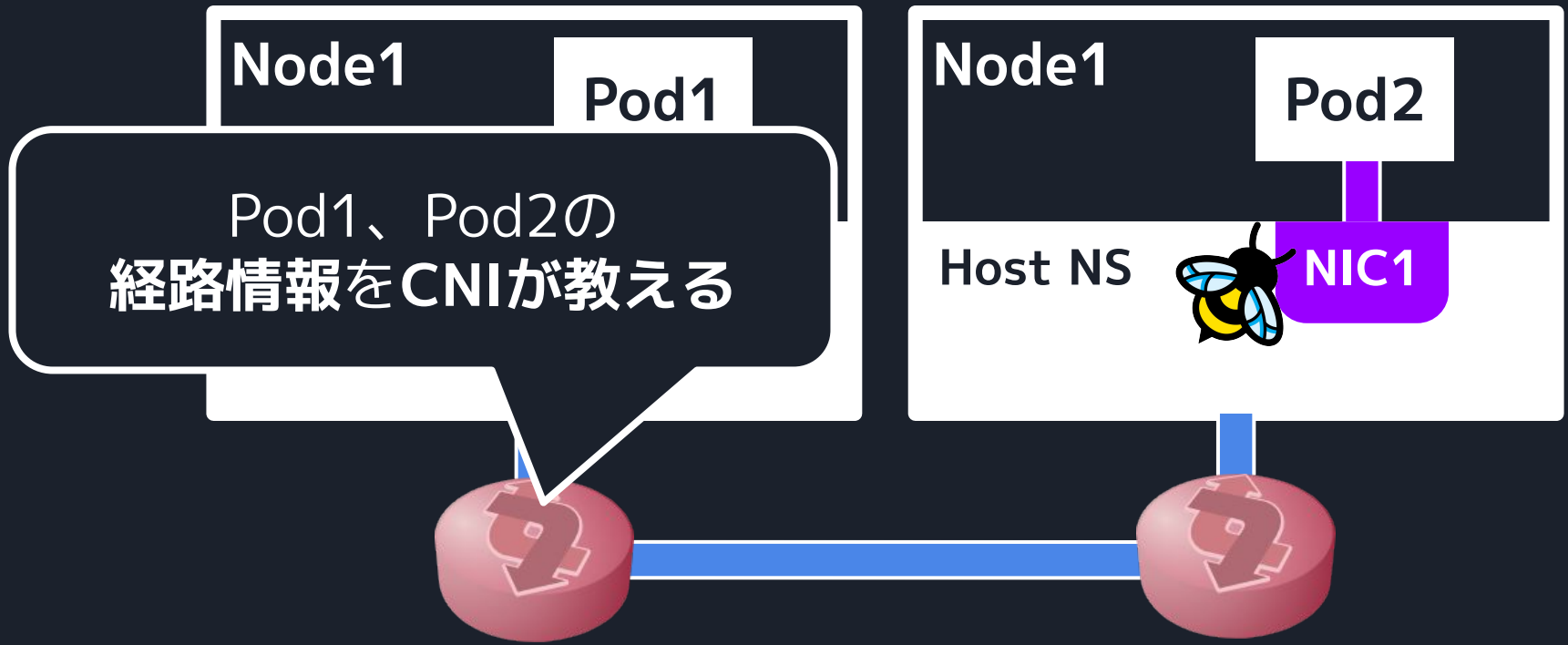
GKE Dataplane V2 - 異なるノード間

Pod作成時: アンダーレイネットワークに設定を流し込む



GKE Dataplane V2 - 異なるノード間

Pod作成時: アンダーレイネットワークに設定を流し込む



GKE Dataplane V2 - 異なるノード間

このような仕組みで
ただパケットを**外に出すだけ**で
勝手に**相手がいるノードに**
届く状態が実現される

Amazon VPC CNI

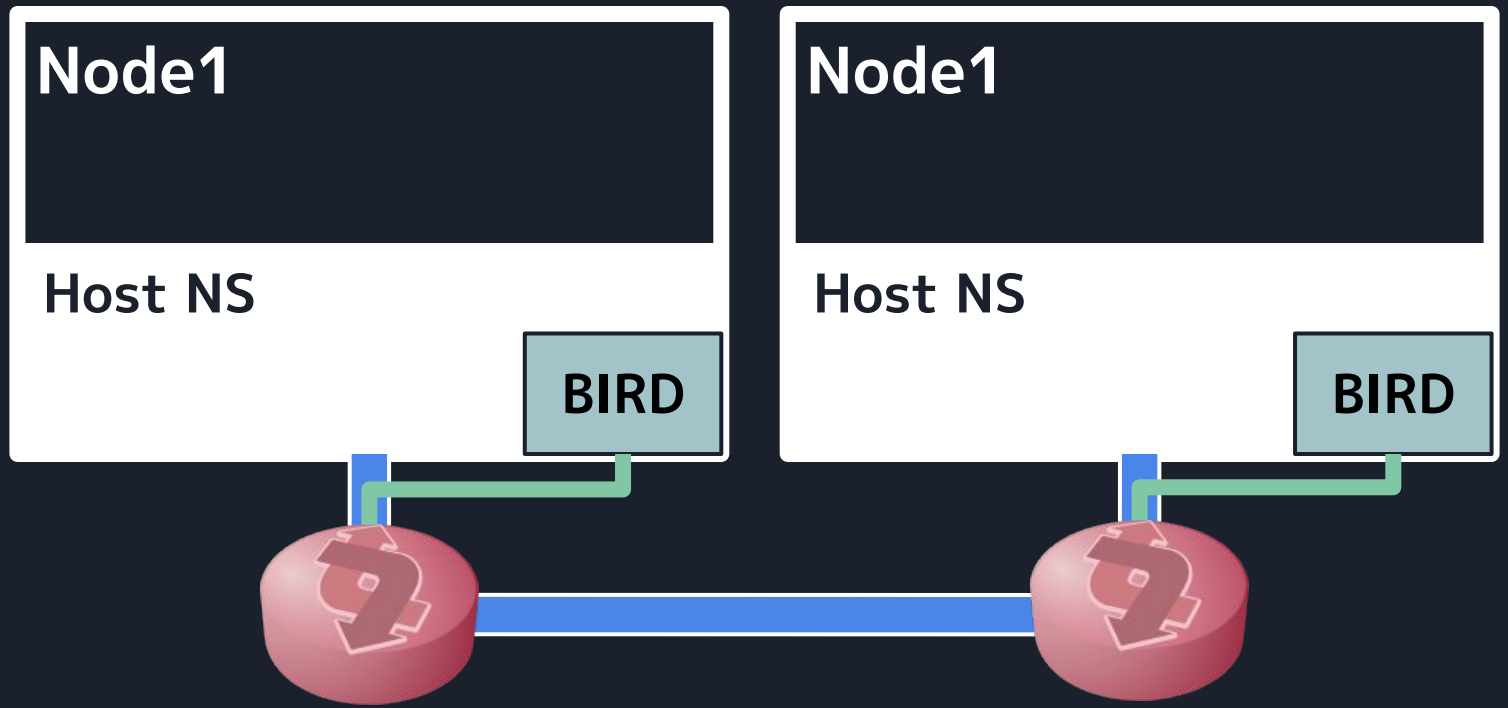
- AWS **VPCとの連携**に非常に強いCNI
- **VPCの中から**PodのIPアドレスが割り当てられる
  - ノードの固定サブネットが無いので、**アドレス空間の効率良い利用**ができる
- 疎通性の確保も**VPCを通して**行われる
  - 詳細はあまり公開されていない
- 同じVPC内にいれば、**VM等と直接通信**ができる

Coil

- CybozuでNeko基盤を作っているチームが開発している**オープンソース**のCNI
- Neko基盤内では、**Cilium** & **BIRD**と組み合わせて運用
- **同じノード内**でのL3疎通
  - **Cilium**の仕組みをそのまま用いる
- **異なるノード間**のL3疎通
  - Calicoと同じくBIRDから**BGPで経路広報**するが、**アンダーレイ**ネットワークに経路を流す

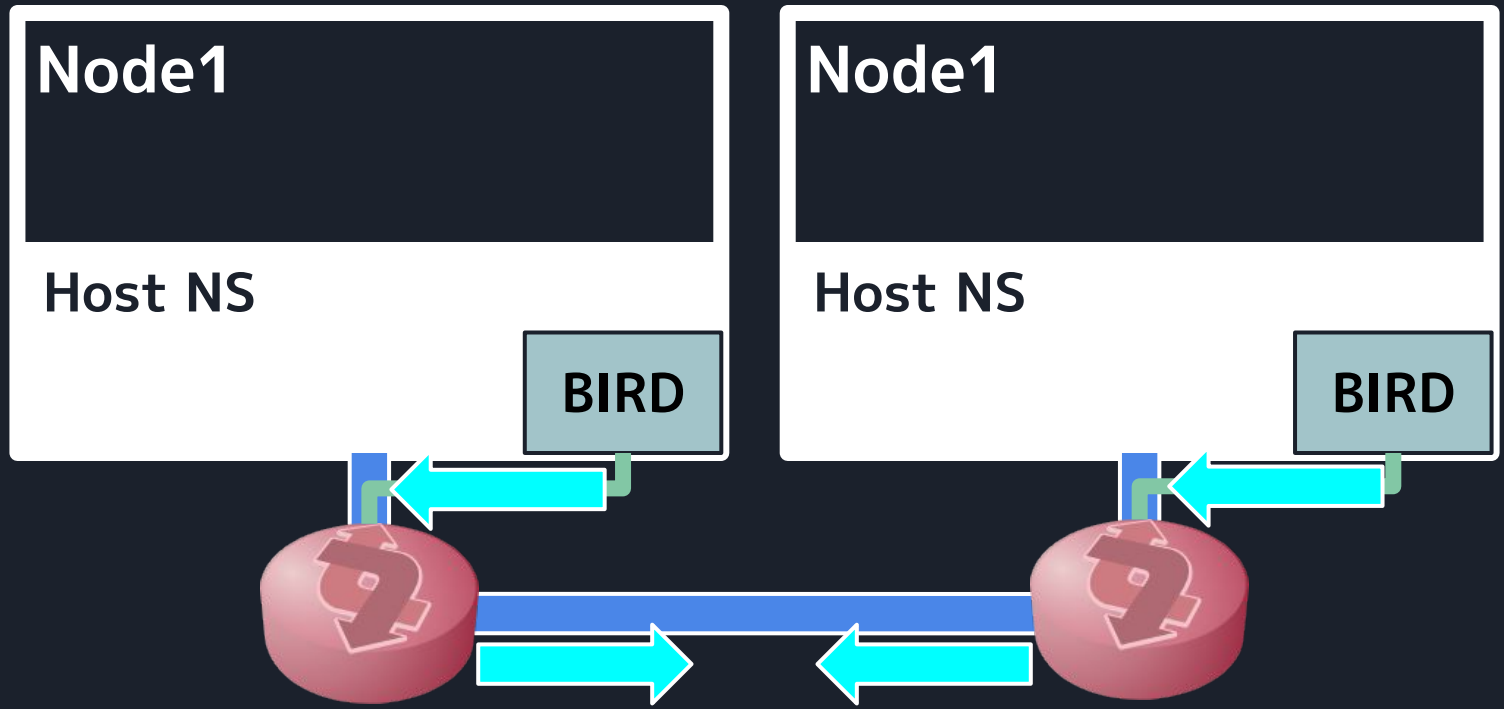
Coil - 異なるノード間

下準備: BIRDで**アンダーレイネットワーク**と**BGPピア**を張る



Coil - 異なるノード間

下準備: **Podサブネット**を全体で互いに**広報**する



CNIはどんな実装も受け入れる
柔軟な思想の表れであり
様々な実装がある

皆さんも馴染みのある技術が多かったのでは

アジェンダ

- イントロダクション
- Kubernetesを知る
- Kubernetesとネットワーク
- Linuxのネットワークスタック
- L2/L3ネットワーク: CNI **←イマココ**
- L4ロードバランサー: kube-proxy
- プラットフォームとの対話で広がる可能性
- まとめ

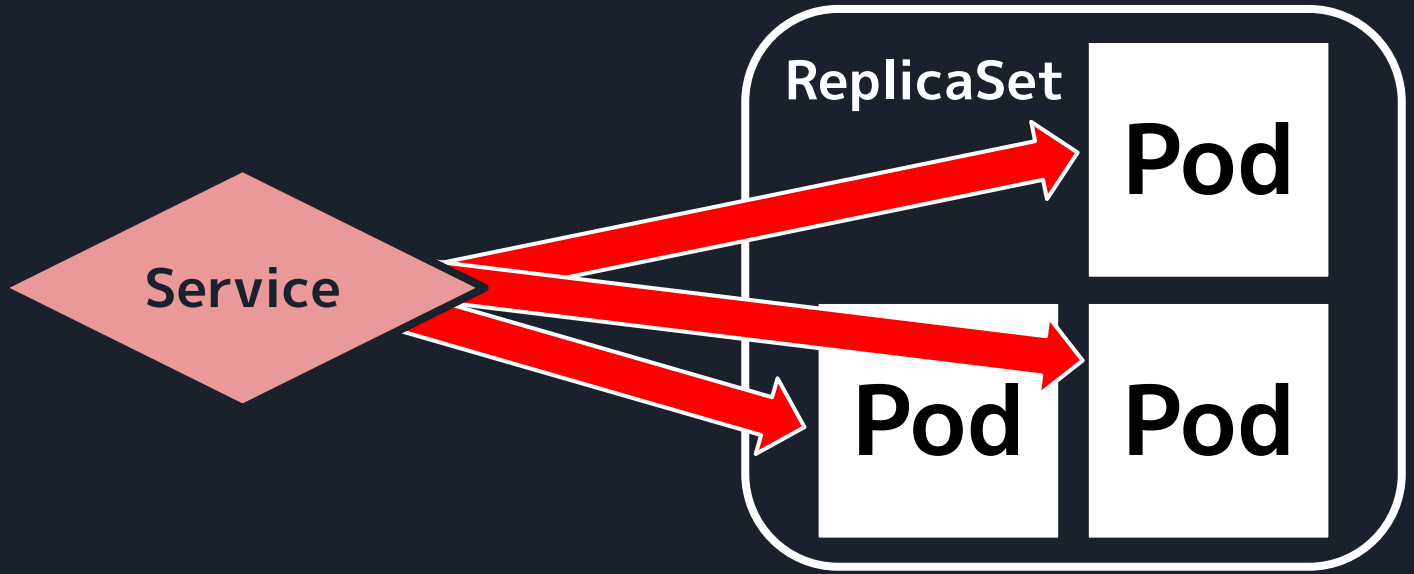
アジェンダ

- イントロダクション
- Kubernetesを知る
- Kubernetesとネットワーク
- Linuxのネットワークスタック
- L2/L3ネットワーク: CNI
- L4ロードバランサー: kube-proxy ←イマココ
- プラットフォームとの対話で広がる可能性
- まとめ

L4ロードバランサー:
kube-proxy

【復習】Service

- 同じ内容のPod群に、均等に**通信を振り分ける** (L4LB)
- Pod群を**1つのIPアドレスでまとめる**ことができる



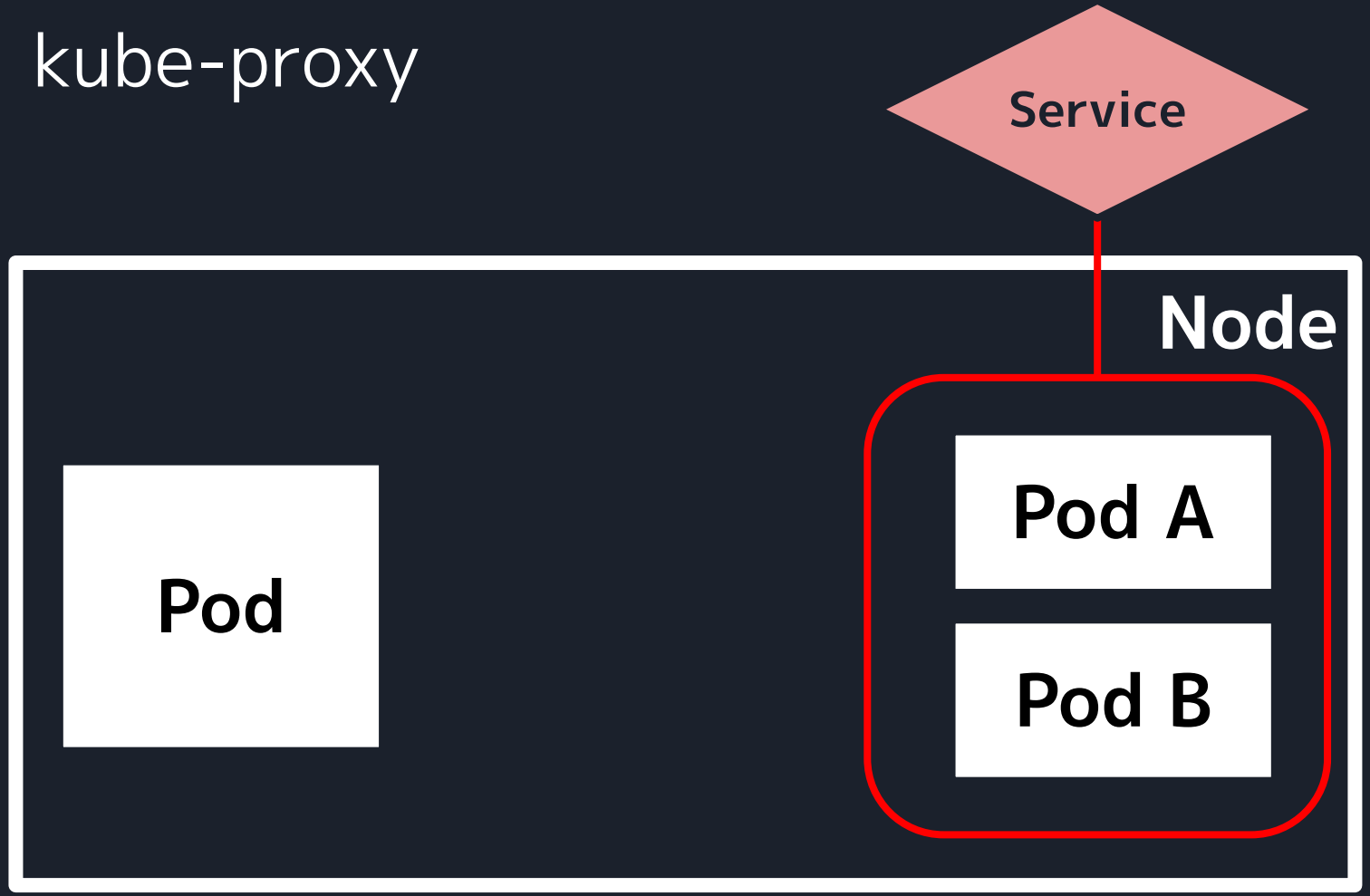
kube-proxy

- **Service**を担当するコンポーネント
- Kubernetesが**デフォルトで持って**いる
- kube-proxyのやる事
  - **宛先**IPアドレスが**Service**の通信があったら、所属する任意PodのIPで**DNAT**する
  - 書き換え先のPodの**バランスをいい感じ**にする

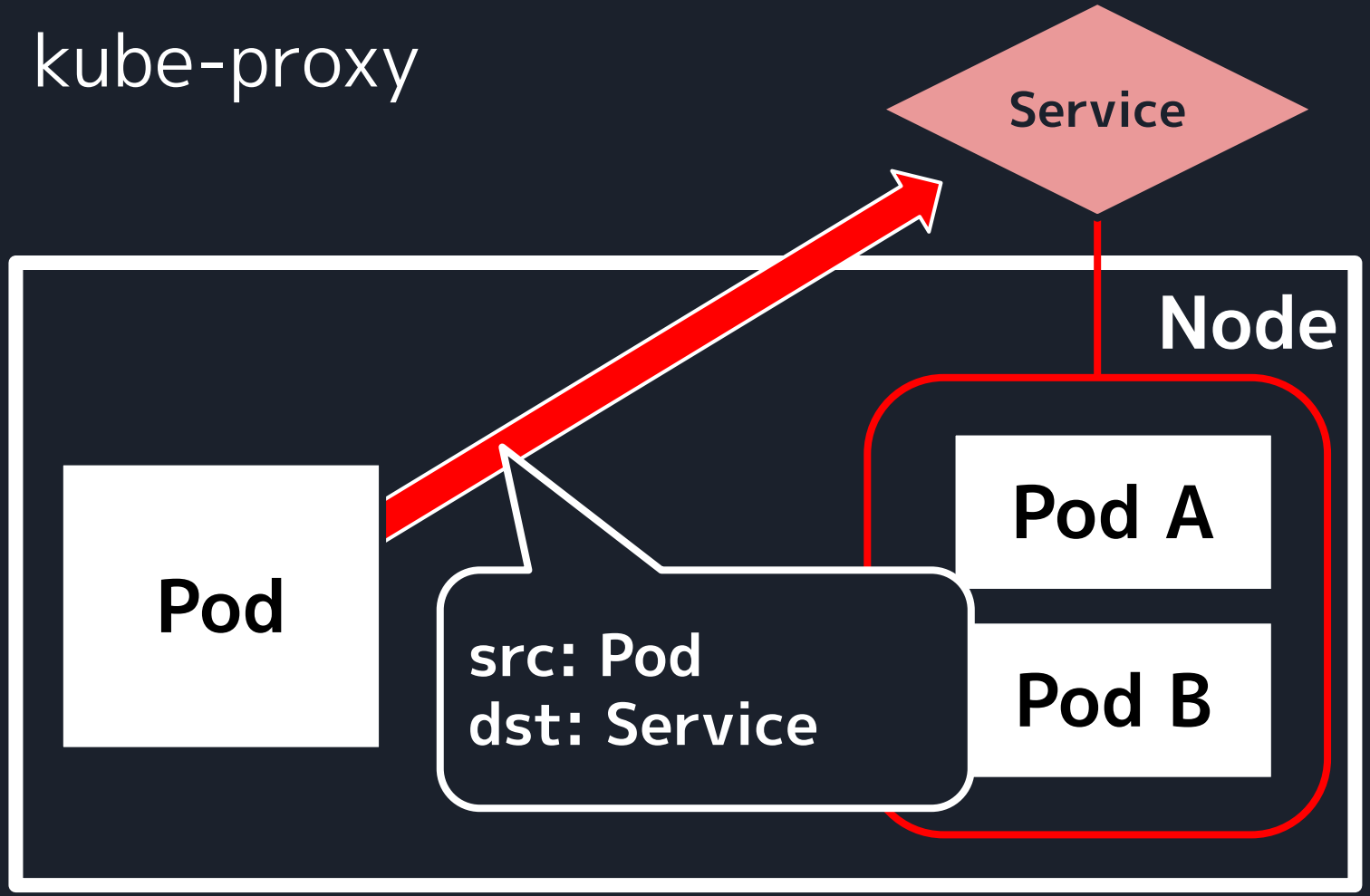
kube-proxyの実装方式

- 基本的には**netfilter**を利用してNATしている
  - kube-proxyは保存されているServiceの定義を監視して**NATルールを流し込む**だけ
  - 実際にプロキシしているわけではない
- 現在のデフォルト: **iptables**を通して設定
- これから: **nftables**への移行が進む (現在ベータ)

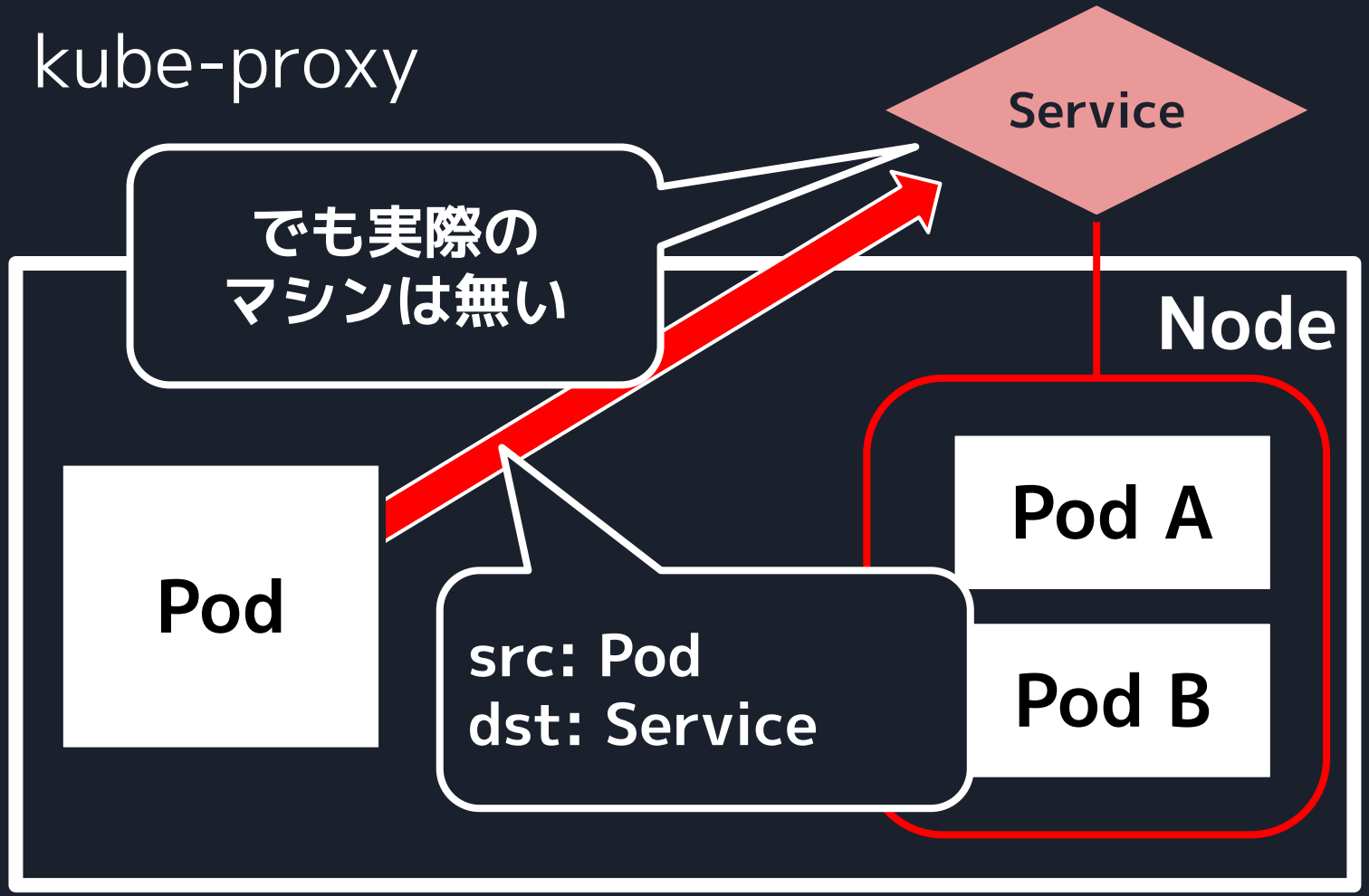
kube-proxy



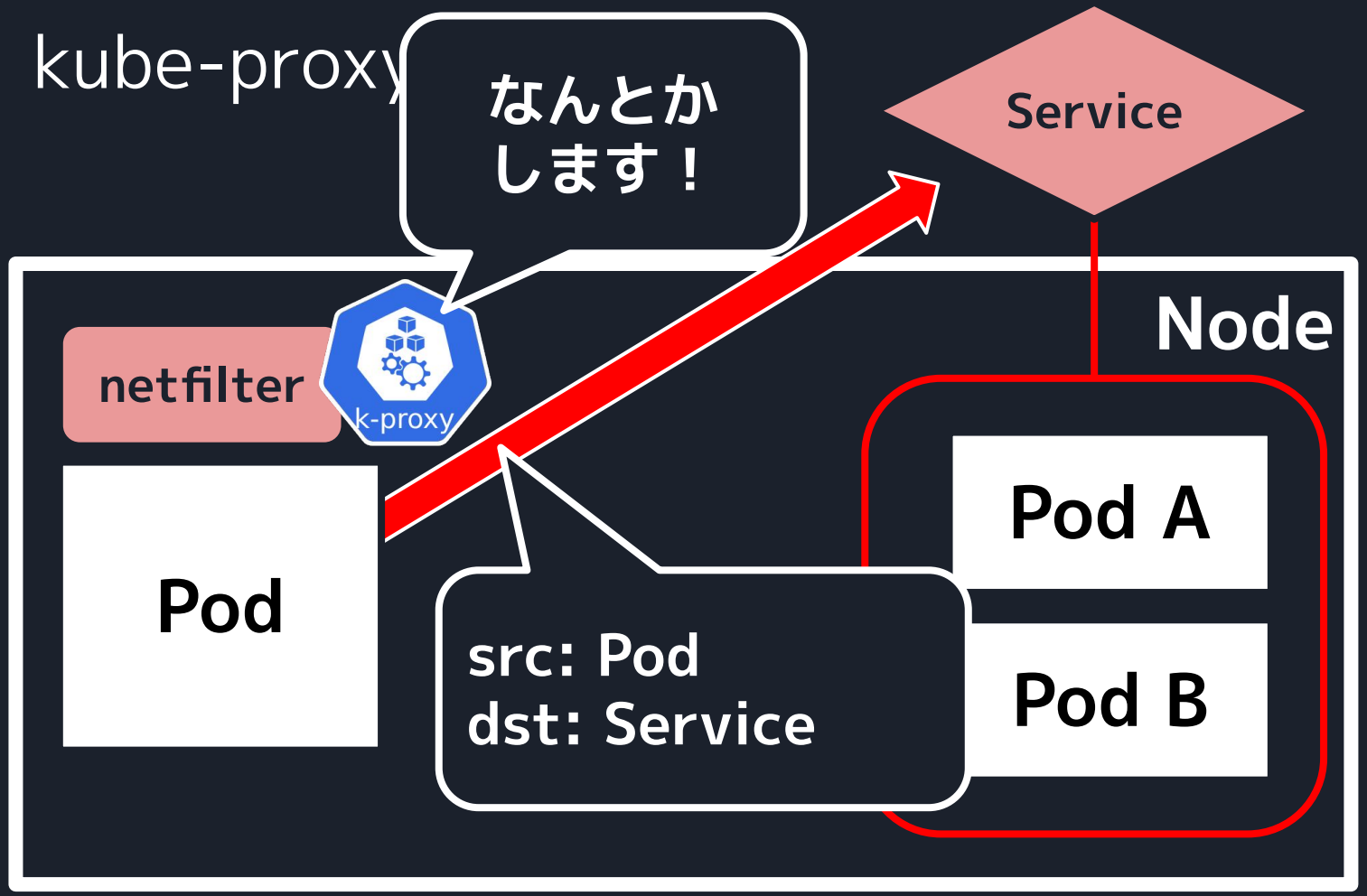
kube-proxy



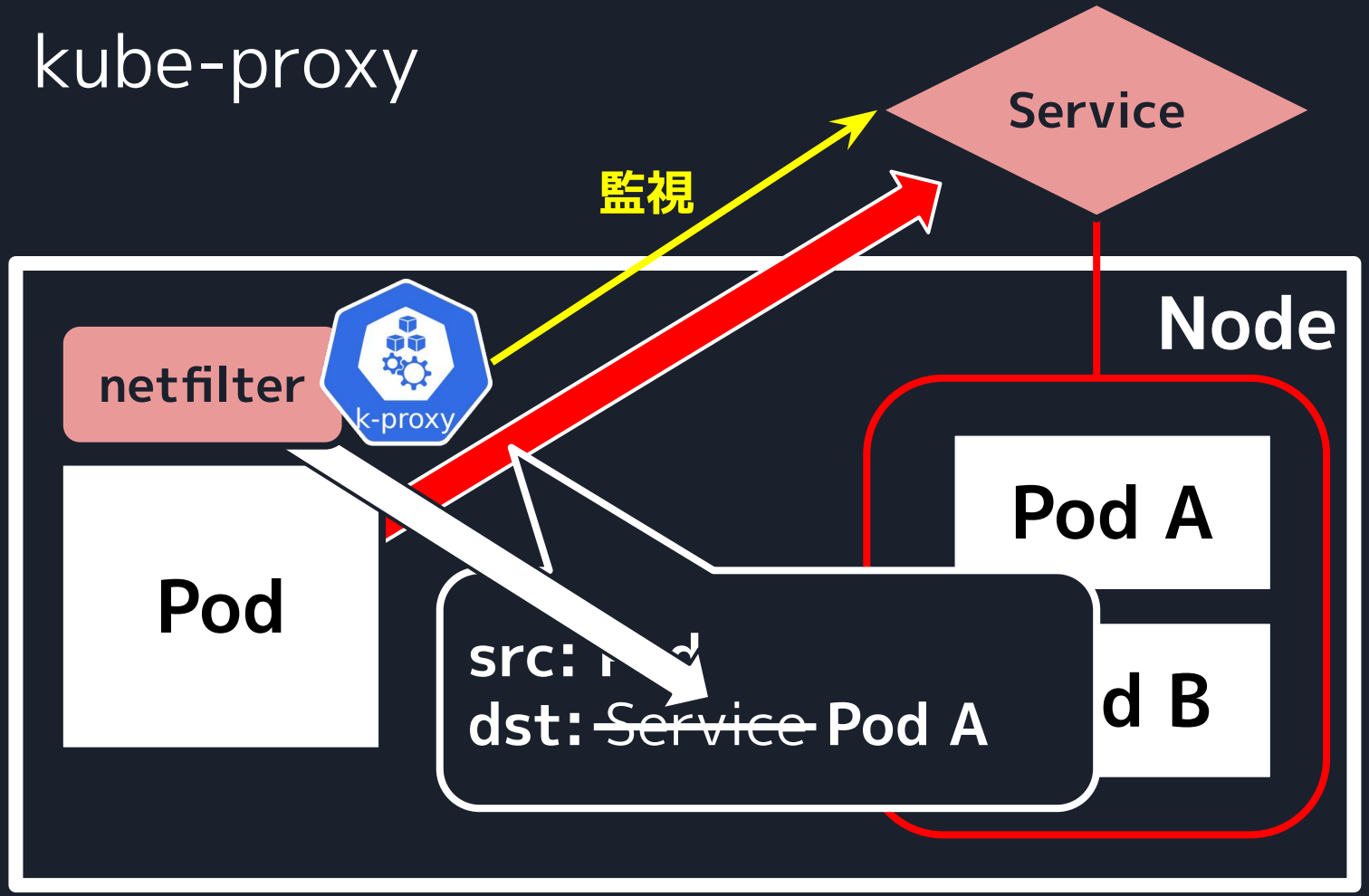
kube-proxy



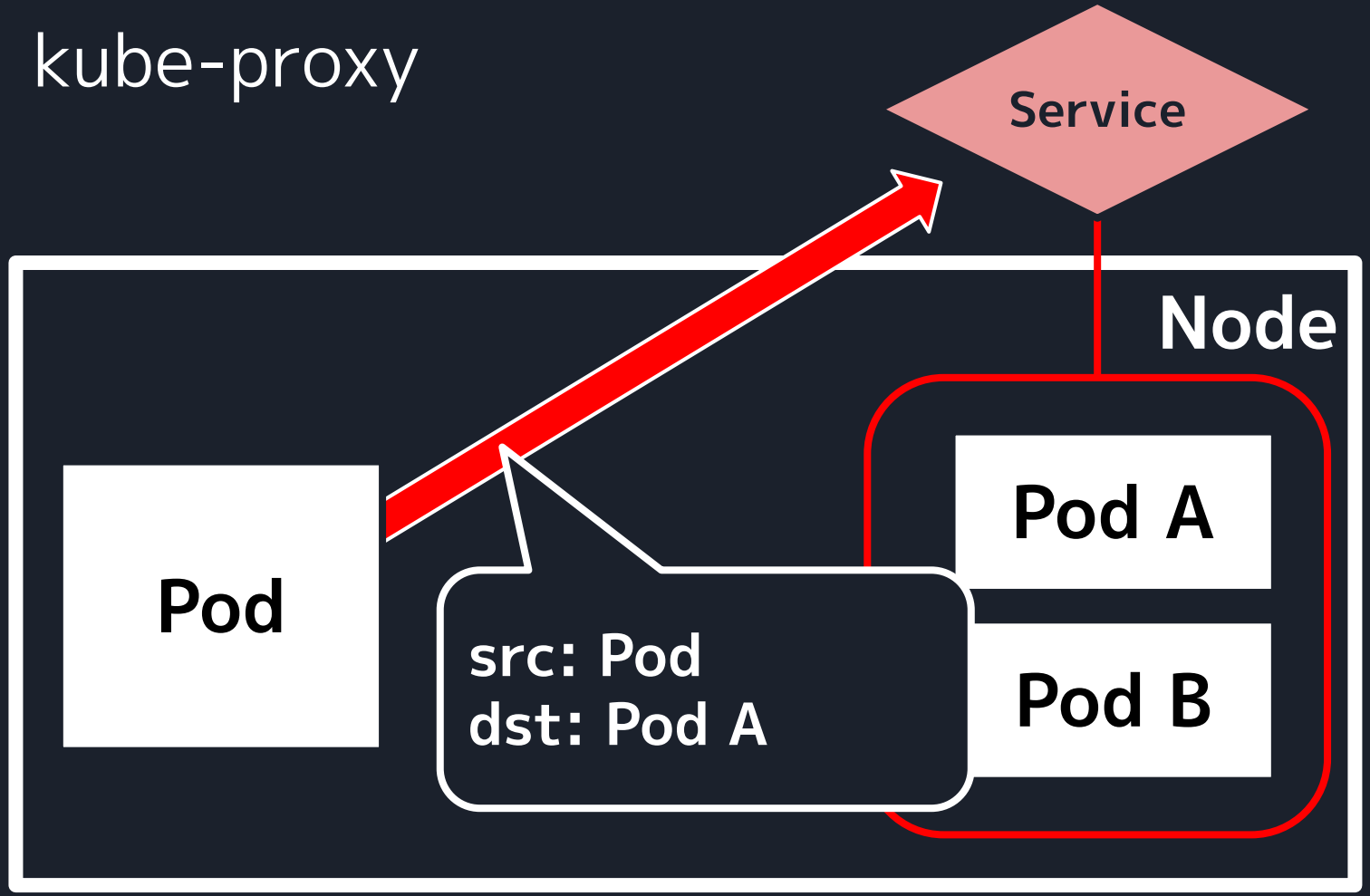
kube-proxy


なんとか
します！
Service
Node
netfilter
k-proxy
Pod
src: Pod
dst: Service
Pod A
Pod B

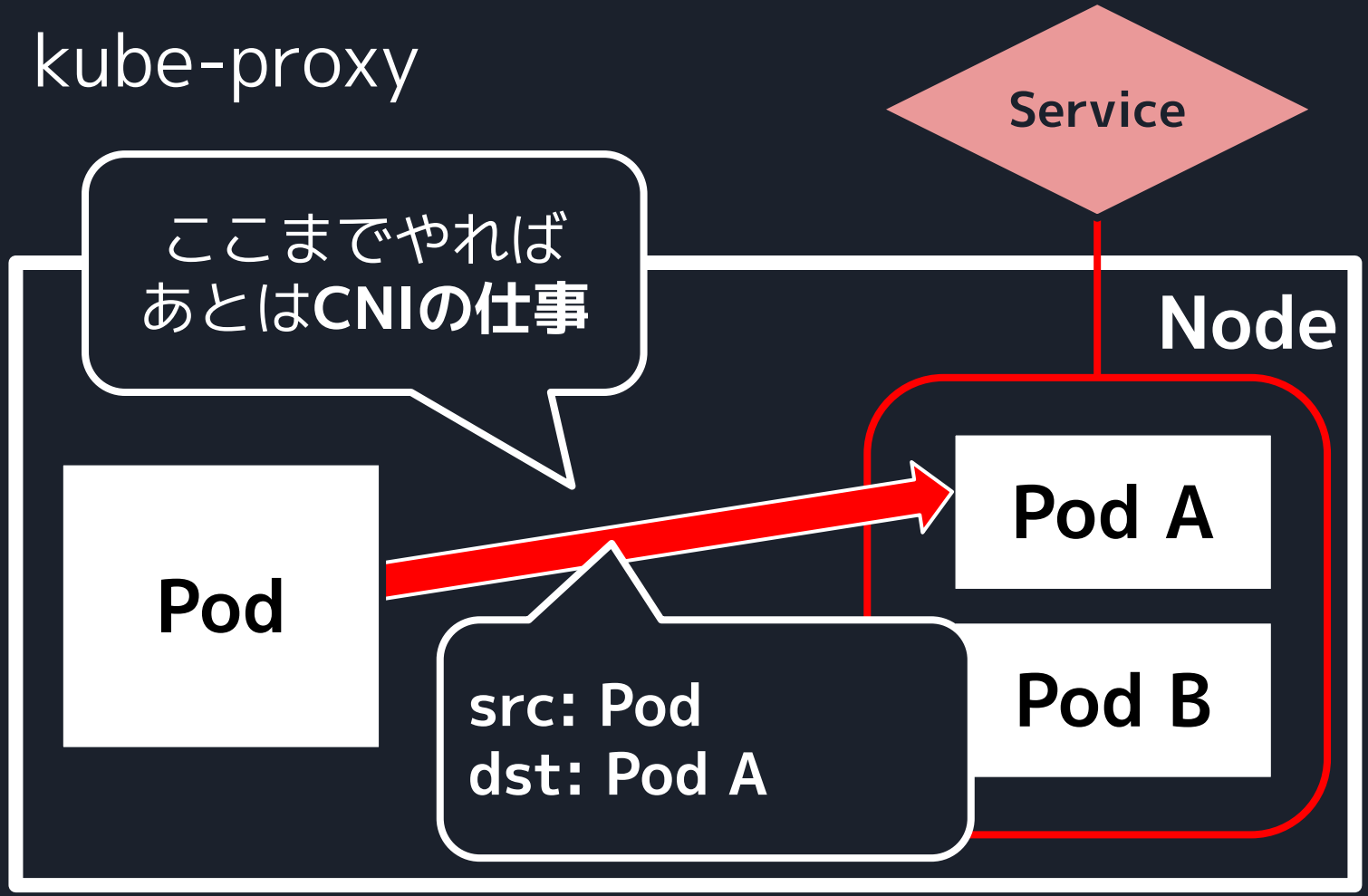
kube-proxy


Service
監視
Node
netfilter
k-proxy
Pod
Pod A
d B
dst: Service Pod A

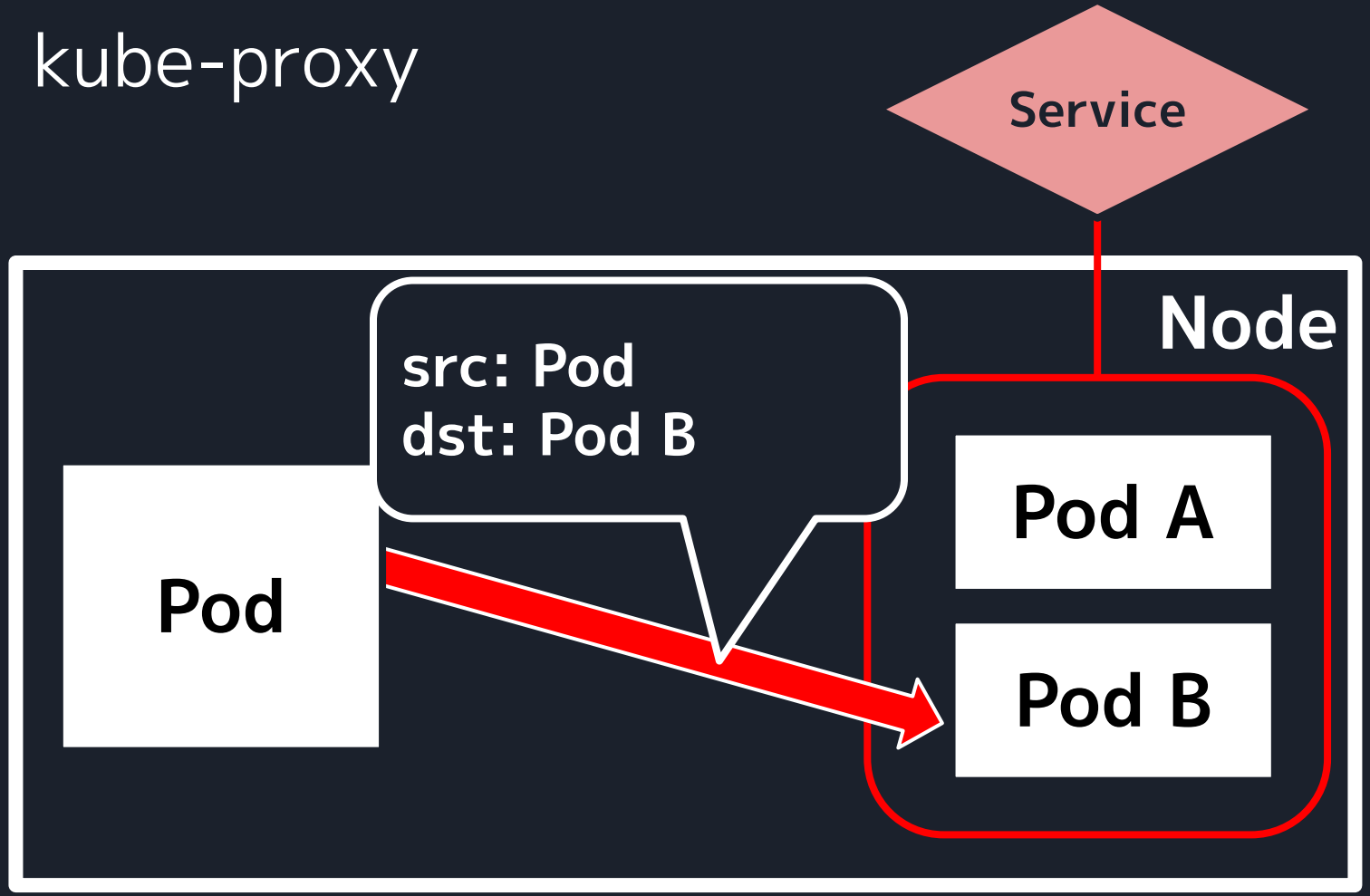
kube-proxy



kube-proxy


Service
ここまでやれば
あとはCNIの仕事
Node
Pod
Pod A
Pod B
src: Pod
dst: Pod A

kube-proxy

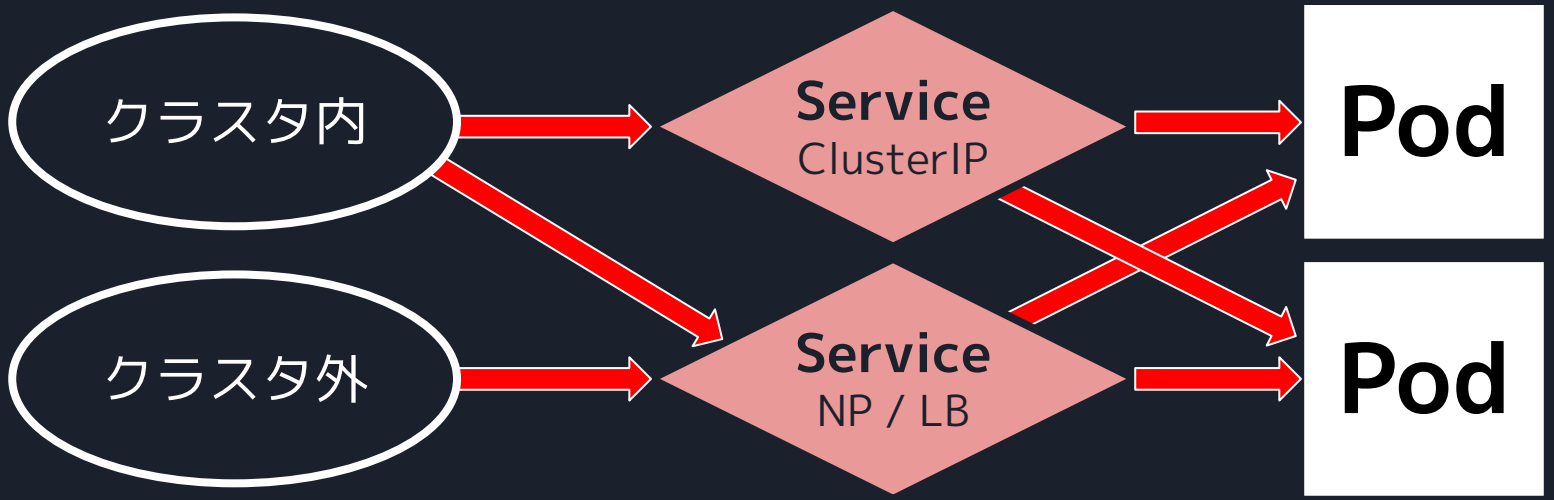


kube-proxy


Service
kube-proxyの役目は
Serviceに属するどこかの
PodにDNATするだけ
Pod B

【復習】Serviceのタイプ

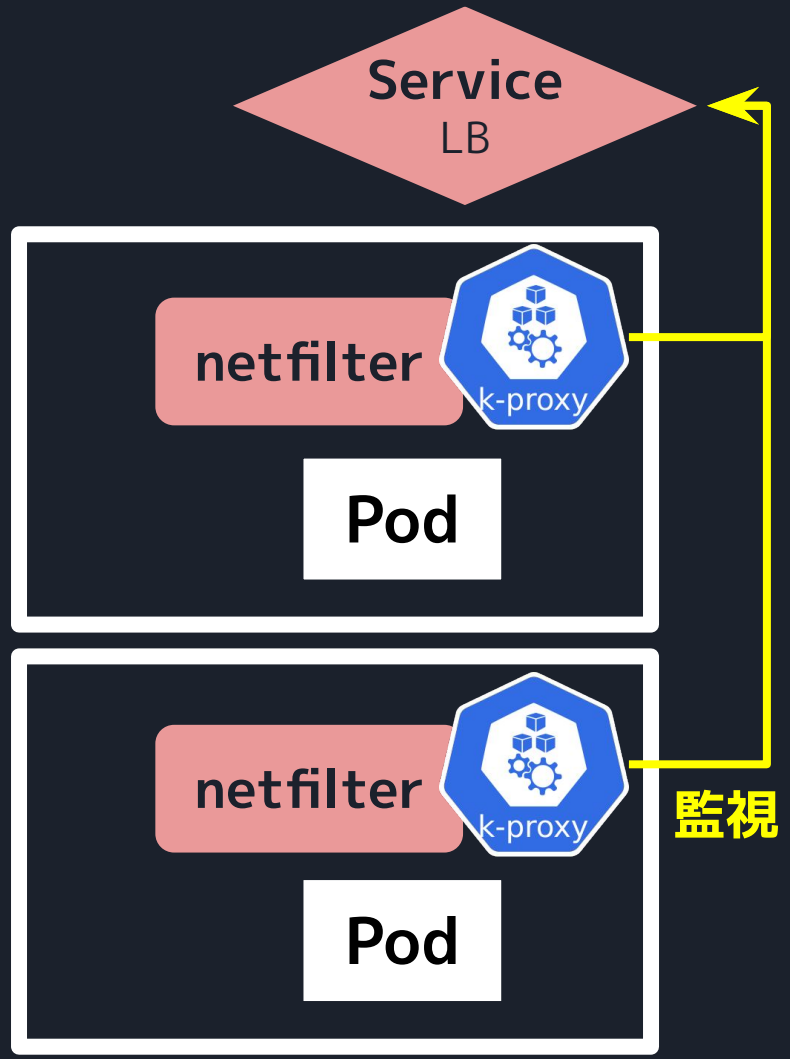
- ClusterIP: **クラスタ内**からの通信のみ受け付ける
- NodePort / LoadBalancer: **クラスタ外**からの通信も受け付ける



外部から通信を受け付けるService

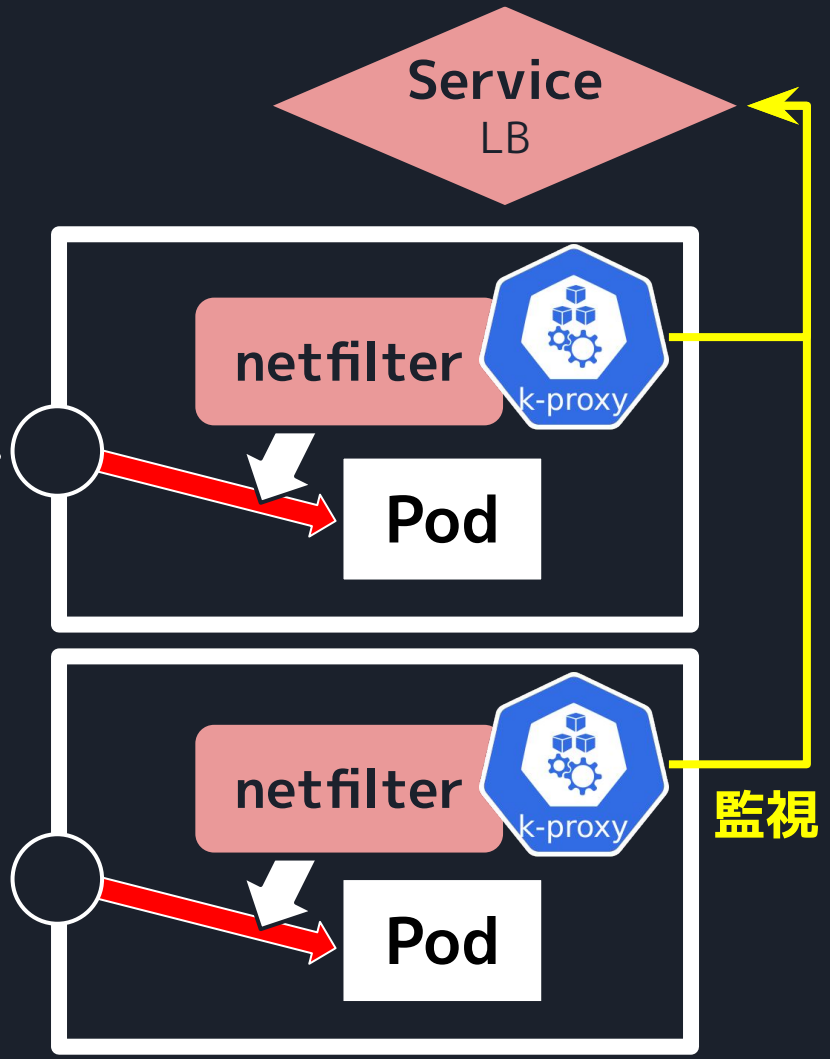
- **NodePort**
  - 30000-32767のうち**1つのポート**を割り当て、**全ノードでそのポート宛**の通信を受け取る
- **LoadBalancer**
  - Nodeportと同じ環境をそろえる
  - **外部L4LB**と連携しクラスタ**外からアクセスできるIPアドレス**を割り当て、LBから ↑ で用意したポートにロードバランシングする

Type: NodePort

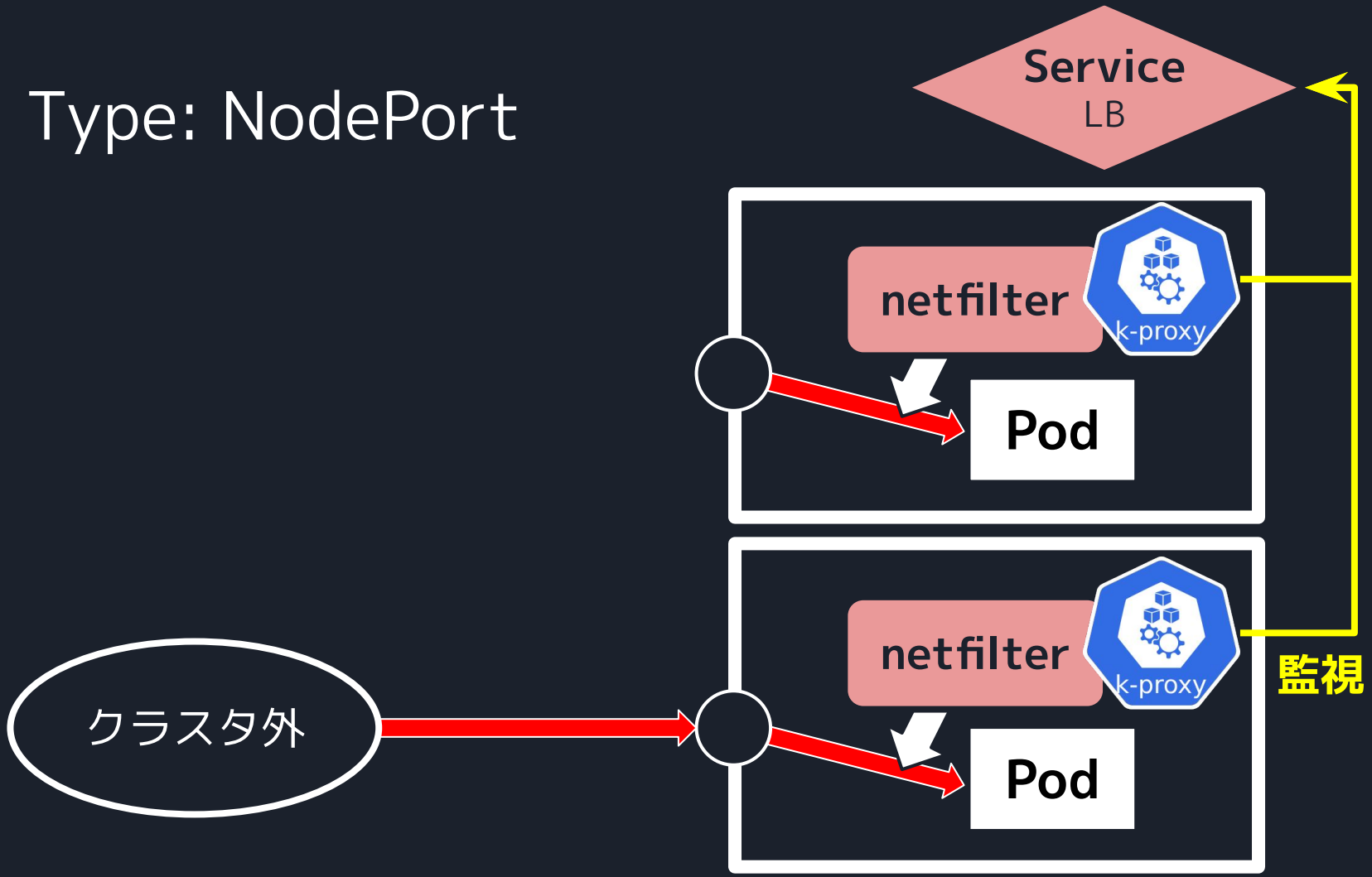


Type: NodePort

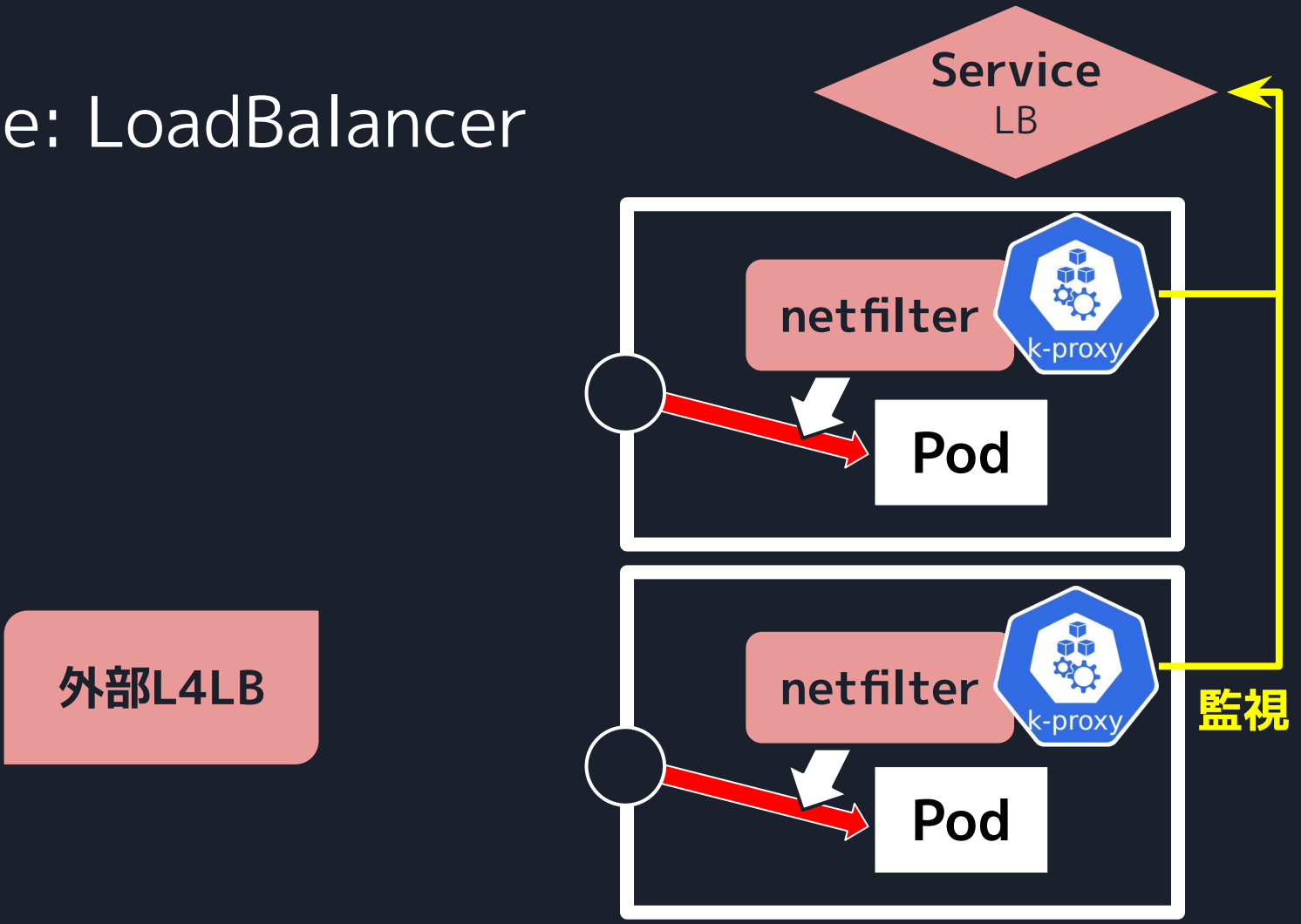
ポートを1つ
割り当て
そこからの**通信を**
キャッチする



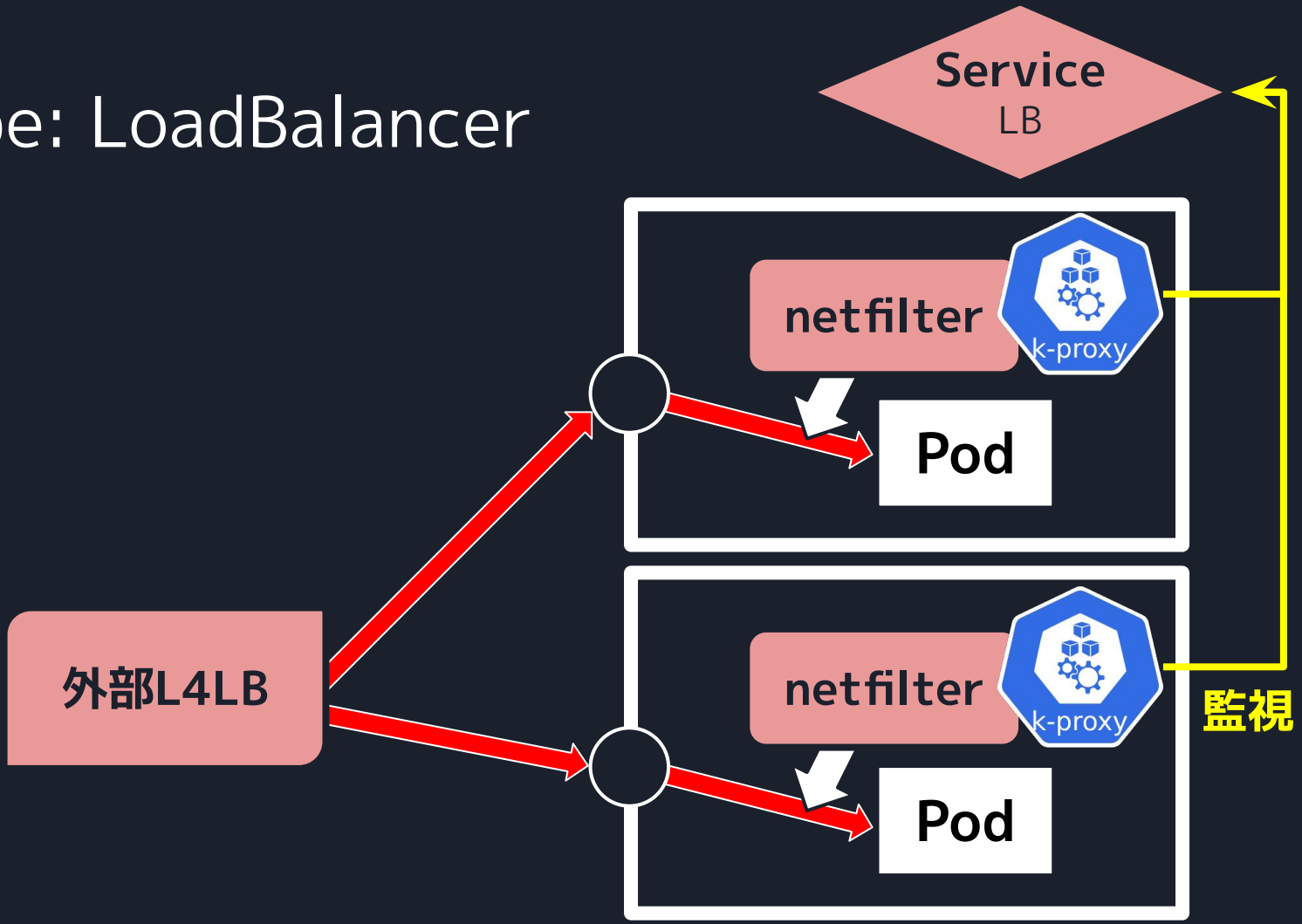
Type: NodePort



Type: LoadBalancer

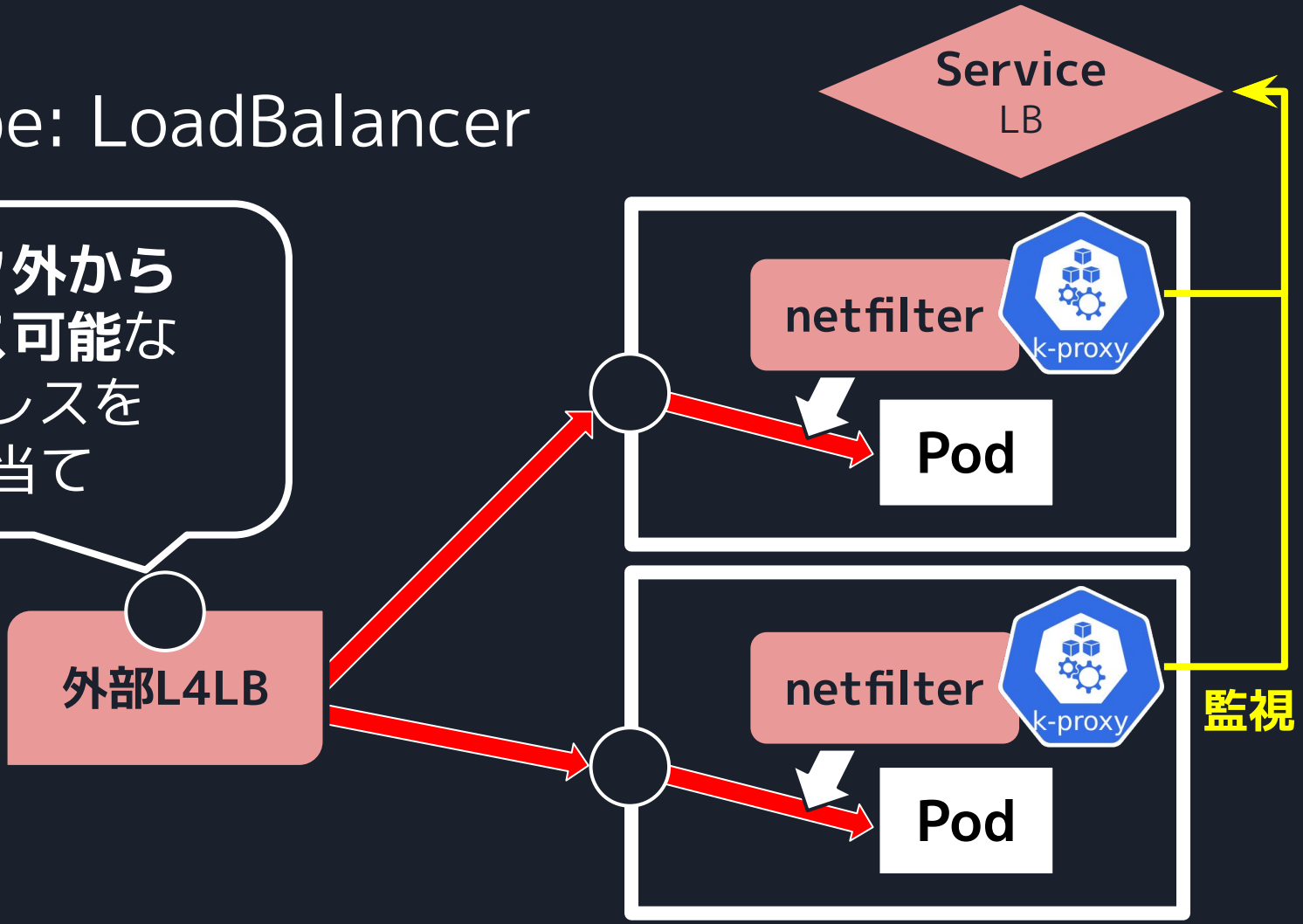


Type: LoadBalancer

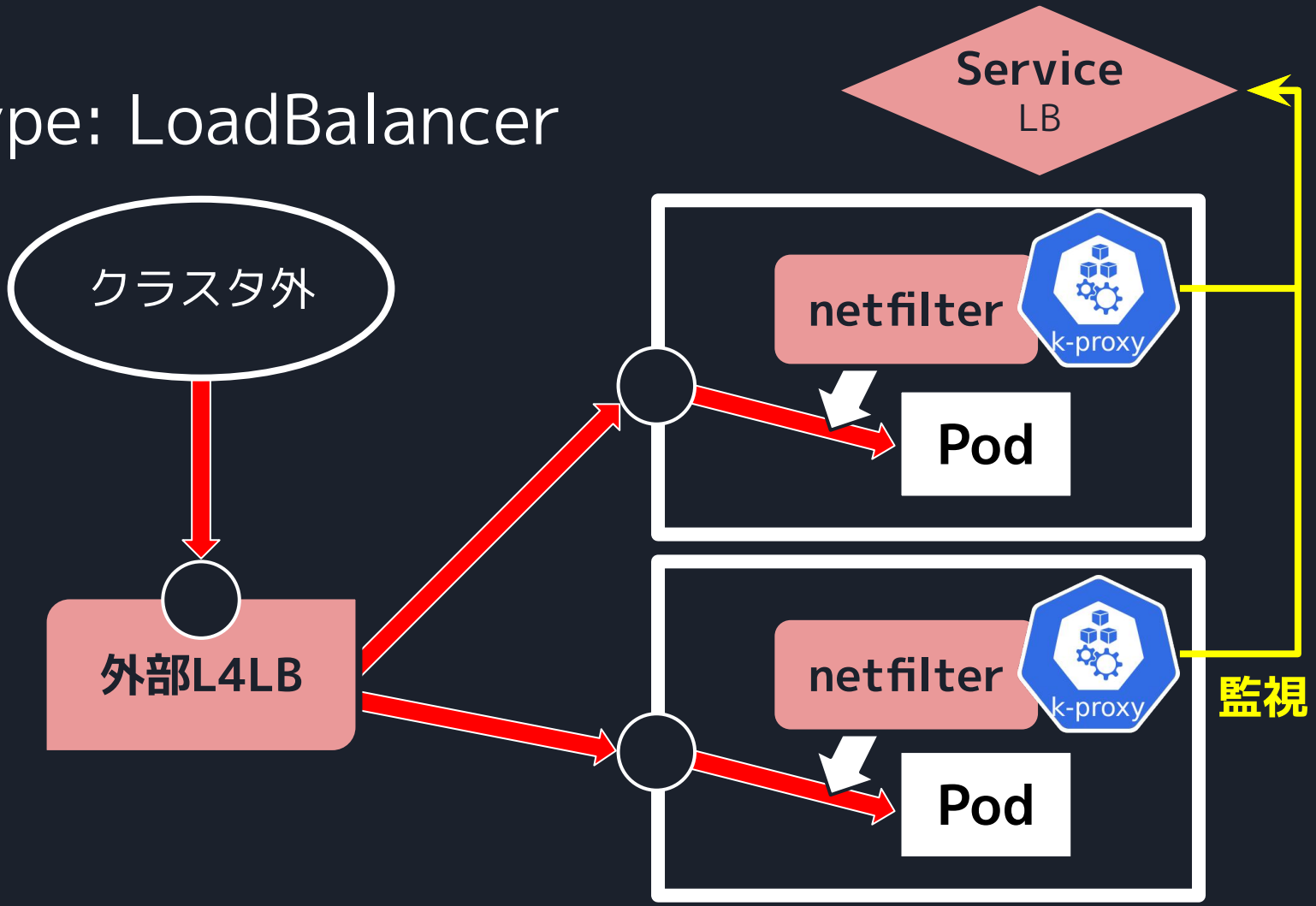


Type: LoadBalancer

クラスタ**外から**
アクセス可能な
IPアドレスを
割り当て



Type: LoadBalancer



外部L4LB

- Type: LoadBalancer用の**外部L4LB**は、Kubernetesがデフォルトで用意して**いない**コンポーネント
  - 既存の基盤に合わせて**選定** or **自作**の必要がある
- 実装例
  - **MetalLB**
    - オンプレ環境を想定した**汎用OSS**
  - **AWS Load Balancer Controller**
    - AWSの**NLB**を使った実装

アジェンダ

- イントロダクション
- Kubernetesを知る
- Kubernetesとネットワーク
- Linuxのネットワークスタック
- L2/L3ネットワーク: CNI
- L4ロードバランサー: kube-proxy ←イマココ
- プラットフォームとの対話で広がる可能性
- まとめ

アジェンダ

- イントロダクション
- Kubernetesを知る
- Kubernetesとネットワーク
- Linuxのネットワークスタック
- L2/L3ネットワーク: CNI
- L4ロードバランサー: kube-proxy
- プラットフォームとの対話で広がる可能性 ←イマココ
- まとめ

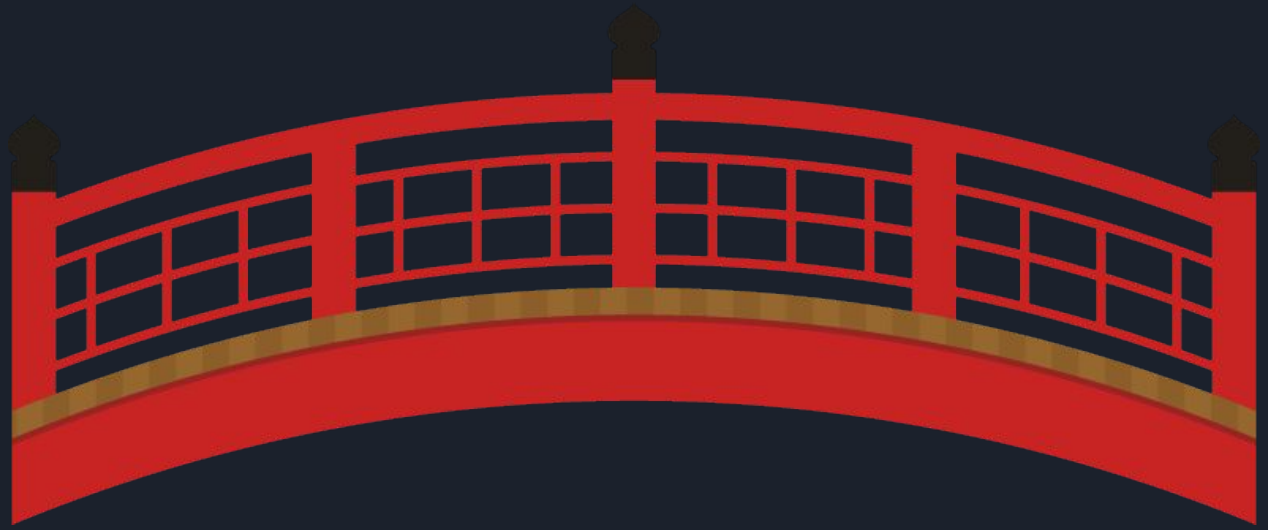
プラットフォームとの
対話で広がる可能性

環境特化なネットワークを選択肢に！

- メジャーなOSSは、どんな所でも使える**汎用性**の代わりに**オーバーヘッド**や**複雑性**を許容している
  - **SDN**はそれなりにノードに処理を要求する
  - **設定項目**も多くなりがち
- K8sのネットワークは**差し替え可能**な部分が多い
  - **自分たちの**ネットワークの状態に合わせて、**最適なものを選ぶ/作る**ことができる！
- この実現のためには、**対話が不可欠**

プラットフォームと対話しよう

- 特にCNIはお互いの**責任を分割**するツール
- ブラックボックスから対話のための**共通言語**へ
- 「どちらも触れない」から「**どちらも触る**」にしたい



実はプラットフォーム方面の活動も…

- 昨年11月にあった、Kubernetes等**クラウドネイティブ**なインフラに関するカンファレンス**CNDW2025**にて「**CNI徹底解説**」というタイトルでNagamiが登壇
- プラットフォームとネットワーク**双方から**「**共通言語**」を広めるのが目標
  - 何か良いアイデアがあればぜひ教えて下さい

まとめ

今日はこんなお話をしました

- イントロダクション
- Kubernetesを知る
- Kubernetesとネットワーク
- Linuxのネットワークスタック
- L2/L3ネットワーク: CNI
- L4ロードバランサー: kube-proxy
- プラットフォームとの対話で広がる可能性
- まとめ

プラットフォームとの
「**フラっと**」な対話で
共にスケールする基盤を！

もっとたくさんの詳細があります、入口になれば幸い

ありがとう
ございました

参考になれば幸いです

【最後に】議論ポイント

- 我々はプラットフォーム側の人間なので、**ネットワークエンジニアの視点**を分かり切れていない
- **ネットワークとプラットフォームはどのように関われるか**について色んな意見が聞きたい
  - ex) PF側・NW側視点でのCNI選定基準
  - ex) CNI改造・自製の判断基準、協力の仕方
  - ex) 責任分界点はどこに置くべきか
  - ex) コンテナネットワークの未来のビジョン