AKS Networking

**Essential Areas to Explore**

* **Exposing Services Externally:**
  + **Ingress:** How to manage external access to your services, including routing based on hostnames and paths, and often used for HTTP/HTTPS traffic. This is crucial for making your applications accessible from the internet.
  + **LoadBalancer Services:** How to provision cloud provider load balancers to expose your services directly via public IPs.
  + **NodePort Services:** Understanding how to expose services on each node's IP at a static port (often used as a building block for Ingress or load balancers).
* **Network Policies:** How to implement network segmentation and control the communication flow between pods within your cluster for security purposes. This allows you to define rules about which pods can talk to each other.
* **Container Network Interface (CNI):** Understanding the underlying networking implementation used in your AKS cluster (e.g., Azure CNI, Calico). While you might not need to know the nitty-gritty details initially, it's helpful for troubleshooting and understanding network behavior.
* **Network Security:** This encompasses various aspects, including:
  + Securing ingress and egress traffic.
  + Using Network Policies for internal security.
  + Potentially integrating with cloud provider security services (like Azure Network Security Groups
* **DNS Configuration:** While Kubernetes manages DNS for internal services, you might need to understand how to configure external DNS for your applications exposed via Ingress or Load Balancers.

**Azure CNI Overlay vs. Azure CNI Node Subnet**

Azure offers different **Container Networking Interface (CNI)** options for **Azure Kubernetes Service (AKS)** to manage pod networking. The key differences between **Azure CNI Overlay** and **Azure CNI Node Subnet** relate to how IP addresses are assigned and how networking is handled to POD.

**1. Azure CNI Overlay (Default for New AKS Clusters)**

Azure CNI Overlay is designed to optimize IP address consumption and improve scalability. It uses an **overlay network** where pods get **virtual IPs** (VxLAN-based) that do not consume IPs from the virtual network (VNet) subnet.

**Key Features:**

✅ **Efficient IP Address Usage** – Pods get IPs from an **internal private range** (100.64.0.0/16 by default) instead of using VNet IPs.  
✅ **Better Scaling** – Supports large-scale clusters with thousands of pods without IP exhaustion.  
✅ **Easier VNet Integration** – No need to allocate large address spaces for pod IPs.  
✅ **Cross-Node Communication via Overlay** – Packets are encapsulated using **VxLAN** and routed across nodes.

**How It Works:**

* Each **pod** gets an IP address from an **internal overlay subnet** (default 100.64.0.0/16).
* The **node** (VM) gets an IP from the **VNet subnet**.
* Pods communicate with each other **via VxLAN tunneling** instead of relying on direct IP routing.

**Best for:**

* Large AKS clusters where VNet IP exhaustion is a concern.
* Deployments that do not require direct pod communication with external services(Azure services)

**2. Azure CNI Node Subnet (Traditional Azure CNI)**

Azure CNI **Node Subnet** (also called **CNI VNet mode**) assigns **real VNet IPs** to **each pod**, making them directly routable within the Azure network.

**Key Features:**

✅ **Pods Get Real VNet IPs** – Pods are assigned IPs from the **same VNet subnet** as nodes.  
✅ **Direct VNet Integration** – No overlay; pods can directly communicate with other Azure resources (e.g., databases, storage accounts) without NAT.  
✅ **Lower Latency** – Since there’s no overlay, packet transmission is direct, reducing network overhead.

**How It Works:**

* Each **node** (VM) gets an **IP from the VNet subnet**.
* Each **pod** gets an **IP from the VNet subnet**, consuming real Azure IPs.
* No encapsulation (VxLAN), so all communication happens directly over the VNet.

**Best for:**

* **Networking-sensitive workloads** (e.g., low-latency applications).
* **Direct communication** with other Azure services without additional routing.
* **Smaller clusters** where IP exhaustion is not a concern.

**Comparison Table**

| **Feature** | **Azure CNI Overlay** | **Azure CNI Node Subnet (Traditional CNI)** |
| --- | --- | --- |
| **Pod IP Assignment** | Internal private range (100.64.0.0/16) | VNet subnet (same as nodes) |
| **IP Consumption** | No real VNet IPs used for pods | Consumes real VNet IPs for each pod |
| **Communication Method** | VxLAN encapsulation (overlay) | Direct routing via VNet |
| **Scalability** | High (supports large clusters) | Limited by available subnet IPs |
| **Performance** | Slight overhead due to VxLAN | Lower latency (direct networking) |
| **Best for** | Large clusters, IP conservation | Low-latency apps, direct pod communication |
|  | | |

**Which One Should You Choose?**

* Use **Azure CNI Overlay** if you need **large-scale AKS clusters** and want to avoid IP exhaustion.
* Use **Azure CNI Node Subnet** if you require **direct VNet integration** and **low-latency networking and need More security for an application.**

**What is My responsibility when configuring the Network Configuration in AKS?**

Yes, **you don’t need to manually assign or manage pod IPs**—Kubernetes and Azure CNI handle it automatically. Here’s how:

**1️⃣ Azure CNI Overlay (VxLAN-based)**

* Pods receive **virtual IPs** from an **internal overlay range** (100.64.0.0/16 by default).
* Kubernetes **assigns these automatically** and uses **VxLAN tunneling** for cross-node communication.
* You don’t configure these IPs manually; Kubernetes and Azure CNI handle it.

**1️⃣ Azure CNI Node Subnet (VNet-based)**

* Pods receive **real VNet IPs** from the **subnet you specify** in Azure.
* Kubernetes **assigns these automatically** based on available IPs in the subnet.
* The only responsibility you have is **ensuring the subnet has enough IPs** to accommodate your expected pod count.

**Your Responsibilities**

🔹 Choose the right CNI mode when creating your AKS cluster.  
🔹 Plan subnet sizes **if using Node Subnet mode** to avoid IP exhaustion.  
🔹 Ensure network policies, NSGs, or firewalls don’t block pod communication.

Other than that, Kubernetes + Azure **handle all IP assignments dynamically!** 🚀

👉 **If you select Azure CNI (Node Subnet Mode), you are responsible for provisioning both the VNet and the subnets—AKS will not automatically create or divide subnets for you.**

**Clarification:**

1️⃣ **If you provide only a VNet and no subnets**, **AKS will fail to deploy**, because it requires explicitly defined subnets.  
2️⃣ **You must create at least one subnet** (typically one for nodes, and optionally another for pods) before creating the cluster.  
3️⃣ **AKS does not automatically create or divide subnets for you**—you must define them in advance and provide them during cluster creation.

I appreciate your keen attention to detail! Let me know if you need further clarification or assistance in subnet planning. 🚀

How will Pods communicate with Outside Overlay Network, when you configured the Azure CNI Overlay?

Azure CNI Overlay

A diagram of a computer

AI-generated content may be incorrect.

Communication with endpoints outside the cluster, such as on-premises and peered VNets, happens using the node IP through Network Address Translation. Azure CNI translates the source IP (overlay IP of the pod) of the traffic to the primary IP address of the VM, which enables the Azure Networking stack to route the traffic to the destination.

When pods need to communicate with services outside of the overlay network (e.g., Azure services, the Internet, or on-premises networks), their private pod IPs (10.10.X.X) are **translated into the node’s IP (192.168.1.4 or 192.168.1.5)** via **NAT**.

**In detail explanation**

This diagram represents the network architecture of a Kubernetes cluster deployed in an Azure Virtual Network (VNet), where a Container Network Interface (CNI) Overlay Plugin is used to enable pod communication across multiple nodes. Let's break it down step by step

**1. Node Architecture**

* The Kubernetes cluster consists of two nodes:
  + **Node1 (192.168.1.4) with a Pod CIDR of 10.10.1.0/24**
  + **Node2 (192.168.1.5) with a Pod CIDR of 10.10.2.0/24**
* These nodes are part of an overlay network, which means the pod-to-pod communication is abstracted and routed through the overlay.
* Each node runs a **CNI Overlay Plugin** to manage the pod-to-pod network communication. This plugin enables inter-node networking using encapsulation (e.g., VXLAN, IP-in-IP, or another tunneling mechanism).
* The nodes are assigned IPs from the **192.168.1.0/24** subnet, which is part of a larger **VNet (192.168.0.0/16).**

**2. Pod Networking**

* Inside each node, the pods are assigned private IPs from the node’s Pod CIDR range:
  + **Node1 Pods:** 
    - Pod 10.10.1.2
    - Pod 10.10.1.3
    - Pod 10.10.1.4
  + **Node2 Pods:** 
    - Pod 10.10.2.2
    - Pod 10.10.2.3
    - Pod 10.10.2.4
* These pod subnets are different from the node’s IP range and need network address translation (NAT) or routing to communicate outside the overlay.

**3. Overlay Network**

* The **Overlay Network** connects the nodes and enables **pod-to-pod** **communication across different nodes.**
* Since each node has a different Pod CIDR, the overlay allows:
  + **Pod 10.10.1.2 on Node1** to talk to **Pod 10.10.2.2 on Node2** as if they were on the same network.
  + The CNI plugin manages this traffic through encapsulation techniques (e.g., VXLAN, IP-in-IP, WireGuard, etc.).
* This overlay network eliminates the need to configure static routes in the underlying network.

**4. Network Address Translation (NAT)**

* When pods need to communicate with services outside of the overlay network (e.g., Azure services, the Internet, or on-premises networks), their private pod IPs (10.10.X.X) are **translated** **into the node’s IP (192.168.1.4 or 192.168.1.5)** via **NAT.**
* The NAT mechanism allows:
  + Outbound communication from pods to Azure services (SQL, Storage, etc.).
  + Access to external resources like on-premises systems or the internet.

**5. Azure Virtual Network (VNet - 192.168.0.0/16)**

* The entire infrastructure resides in an Azure Virtual Network (VNet), which allows secure, isolated networking.
* The **192.168.1.0/24** **subnet** is a part of this VNet, hosting the Kubernetes nodes.
* This VNet can be **peered with on-premises networks** or connected to other VNets.

**6. External Communication**

* The nodes are connected to:
  + Azure Services (e.g., Azure SQL, Storage)
  + The Internet for external API calls or updates
  + On-premises infrastructure via VPN or ExpressRoute

Key Takeaways

1. **CNI Overlay Plugin** creates an overlay network to enable pod-to-pod communication across multiple nodes.
2. **NAT** allows pods to communicate with external networks by translating their IPs into the node’s IP.
3. **Azure VNet Integration** ensures the Kubernetes cluster can securely communicate with Azure services, the internet, and on-premises resources.
4. **Encapsulation** (via VXLAN, IP-in-IP, etc.) allows seamless pod networking across different subnets without requiring direct routing.

Would you like a breakdown of how to configure this setup in Azure Kubernetes Service (AKS)?

How will Pods communicate with Outside Overlay Network, when you configured the Azure CNI Overlay?

Azure CNI Node Subnet

A diagram of a cloud computing network

AI-generated content may be incorrect.

This image represents the **Azure Container Networking Interface (CNI)** architecture for Kubernetes networking. It details how **pods, nodes, and virtual network components** are interconnected within an Azure **Virtual Network (VNET)** using an **L2 bridge-based networking model**.

**Breakdown of the Image Components:**

**1. Virtual Network (VNET) and Subnet:**

* The entire Kubernetes cluster is deployed within an **Azure Virtual Network (VNET)** (10.0.0.0/16).
* Inside the VNET, a **subnet (10.0.0.0/24)** is allocated for **Azure Kubernetes Service (AKS) nodes and pods**.

**2. Nodes and Pods:**

* **Node 1** and **Node 2** represent Kubernetes worker nodes running inside the AKS cluster.
* Each node has **multiple pods** running within it.
  + **Node 1 hosts:**
    - **Pod 1** with **Container 1** and **Container 2** (eth0 with 10.0.0.101/24).
    - **Pod 2** with **Container 1** (eth0 with 10.0.0.102/24).
  + **Node 2 hosts:**
    - **Pod 3** with **Container 1** (eth0 with 10.0.0.203/24).

**3. Azure CNI Networking:**

* **Each pod gets an IP address from the Azure VNET subnet (10.0.0.0/24)**, which means they are directly routable within the VNET.
* Unlike other CNIs (like Flannel or Calico) that use overlay networks, Azure CNI assigns real **IP addresses to pods from the subnet**, ensuring better performance and security.
* Each pod has a **virtual Ethernet interface (eth0)** attached to an **Azure bridge (azure0)**.

**4. L2 Network and Bridge:**

* The **azure0 bridge** is responsible for connecting all pods to the underlying Azure network.
* The **L2 network** (depicted by a cloud) allows all network interfaces to **exist in a shared broadcast domain**, enabling direct communication without requiring NAT or encapsulation.
* The **bridge (azure0)** acts as a local switch, ensuring that the pod traffic remains within the node whenever possible.
* The **azv interfaces** represent virtual network interfaces that connect the containers to the bridge.

**5. Inter-Node Communication:**

* **Pods across different nodes communicate via the Azure network fabric**.
* The **azure0 bridge on each node** is responsible for ensuring traffic is properly forwarded between the **L2 domain (same subnet)**.
* Azure handles **routing between different nodes**, so pods on **Node 1 (10.0.0.101)** can directly communicate with pods on **Node 2 (10.0.0.203)** via the VNET.
  1. **Communication**
* Since **Azure CNI assigns Pod IPs from the VNet, Pods on different nodes communicate directly** using **Azure's routing tables**—no overlay or VXLAN tunneling is required**.**
* In the **Azure CNI overlay model (image above),** pod-to-pod traffic would require encapsulation (VXLAN..etc).
* Pod-to-External Communication (Azure SQL In another subnet**), Azure Handles routing automatically** since the **pod’s IP is a part of the VNet.**

**Key Takeaways from the Diagram:**

1. **Azure CNI assigns VNET IPs to pods**: Unlike traditional CNIs, Azure CNI allows pods to receive **real IPs from the Azure Virtual Network**.
2. **Direct routing**: Since pods use the VNET subnet, no encapsulation (like VXLAN or IP-in-IP) is needed.
3. **L2 bridge networking**: Nodes use a **Linux bridge (azure0)** to interconnect pods within the same node.
4. **Efficient cross-node communication**: The Azure network fabric handles pod-to-pod communication across different nodes.
5. **No need for NAT**: Since pods get real subnet IPs, they can directly communicate with other Azure services (like databases, storage, and APIs) without NAT.

Would you like me to elaborate on any specific part?

What is Service?

Service enable communication between different components of an application, or between applications, in a stable and reliable way.

explain how communication happens between your microservices using DNS within the same node pool in your AKS cluster?

To explain how communication happens between your microservices using DNS within the same node pool in your AKS cluster, here's a breakdown:

When you create a ClusterIP service in Kubernetes, it gets assigned a DNS name by the kube-dns (or CoreDNS, depending on your AKS version) service running in the cluster. This allows your microservices to discover and communicate with each other without needing to know the individual IP addresses of the pods.

Here’s how it works:

1. **Service Creation:** You've created two ClusterIP services. Let's say one is named service-a and the other is named service-b. If they are in the default namespace, their DNS names will be:
   * **service-a.default.svc.cluster.local**
   * **service-b.default.svc.cluster.local**
2. **DNS Resolution:** When a pod in **service-a** needs to communicate with **service-b,** it will make a DNS query for **service-b.**
3. **Kube-DNS/CoreDNS:** The kube-dns (or CoreDNS) service intercepts this DNS query. It maintains a record of all the services in the cluster and their corresponding ClusterIP addresses.
4. **IP Address Return:** Kube-dns resolves **service-b** to its ClusterIP address.
5. **Traffic Routing (kube-proxy):** The traffic from the pod in **service-a** will then be routed to the ClusterIP of **service-b.** Kubernetes **kube-proxy** (running on each node) will then take over and forward the traffic to one of the pods backing **service-b** based on the service's load-balancing policy.

A screenshot of a computer program

AI-generated content may be incorrect.

**In summary:** Your microservices within the same node pool can communicate using the DNS names assigned to their respective ClusterIP services. Kubernetes' internal DNS service (kube-dns or CoreDNS) handles the name resolution to the ClusterIP, and kube-proxy takes care of routing the traffic to the actual pods.

ok, what if microservices are in different node pool?

If your microservices are in different node pools within the same AKS cluster, the communication using DNS and ClusterIP services will still work seamlessly.

A diagram of a computer

AI-generated content may be incorrect.

**DNS Resolution Unchanged:** When a pod needs to communicate with another service using its DNS name **(e.g., service-b.default.svc.cluster.local),** the kube-dns (or CoreDNS) service will still resolve this name to the ClusterIP address of **service-b**. The location of the pods backing service-b (in this case, in a different node pool) is transparent to the DNS resolution process.

**Key takeaway:** The location of your microservice pods within different node pools does not affect how they communicate using ClusterIP services and DNS. Kubernetes handles the underlying networking to ensure that traffic reaches the correct service endpoints.

What if microservices are in 2 different clusters?

* If your microservices are in 2 *different* Kubernetes clusters, the communication becomes more complex because ClusterIP services are inherently private to their own cluster. They are not directly reachable from outside the cluster.
* To enable the communication between microservices are in two different clusters, you’ll need to expose the service in one cluster then the other cluster can access them. Here are some common approaches:

1. **Exposing Services with LoadBalancer or NodePort:**

* **LoadBalancer:** You can change the service type from ClusterIP to LoadBalancer. This will provision an external load balancer (Provide by your cloud provider) that exposes the service to the external world via a public IP address. The other cluster can then access the service using this external IP address and the specified port.
* **NodePort:** You can change the service type to **NodePort.** This exposes the service on a specific port on all the nodes in the cluster. The other cluster can then access the service using the public IP address of any node in the first cluster and the **NodePort**. You might typically put a cloud provider load balancer in front of the **NodePort** services for better load balancing and reliability**.**

**2. Using Ingress Controllers:**

* You can use an Ingress controller in the cluster where the service is running. Ingress can expose multiple services through a single external IP address and manage routing based on hostnames or paths. The other cluster can then access the services via the Ingress controller's external endpoint.

**Benefit:** Provides more control over routing and can handle TLS termination.

**3. Utilizing a Service Mesh with Multi-Cluster Support:**

* Service meshes like Istio or Linkerd offer features specifically designed for multi-cluster communication. They can handle service discovery, secure communication (mTLS), traffic management, and observability across multiple clusters. This is a more advanced but often preferred solution for complex multi-cluster architectures**.**

**4. Establishing a VPN or Private Network Peering:**

* You can set up a secure connection between the virtual networks of your two Kubernetes clusters using a VPN or private network peering. Once the networks are connected, the services in one cluster might be able to reach the internal IP addresses (including ClusterIPs if configured correctly with appropriate routing) of the services in the other cluster. This often requires more complex network configuration.

**Drawback:** These methods expose your services publicly, which might not be desirable for all inter-service communication. You'll need to manage security carefully.

As you mentioned About CoreDNS, so may it have One Private IP? And will it be from the Component called Kubernetes DNS service IP address (while you entered Networking in AKS)?

A screenshot of a computer

AI-generated content may be incorrect.

Yes, you are on the right track! When you create an AKS cluster, Kubernetes assigns a clusterIP address to the kube-dns (or CoreDNS) service This IP address is the one that pods within your cluster use to resolve DNS queries. This IP address is typically within the service IP range you defined during cluster creation.

So, yes, in essence, there will be a single IP address for the CoreDNS service that your microservices will use for DNS resolution.

Now, regarding whether a single IP address is "good" for resolving DNS queries between multiple services, the answer is generally **yes, it is good and is the standard way Kubernetes handles internal DNS resolution.** Here's why:

* **It's a Service IP:** The IP address assigned to CoreDNS is a ClusterIP, which is an internal virtual IP managed by Kubernetes. This IP isn't tied to a specific pod.
* **Load Balancing Behind the Scenes:** The CoreDNS service is typically backed by multiple pods running in your cluster. Kubernetes uses its internal load balancing mechanisms (managed by kube-proxy or similar components) to distribute DNS queries sent to the CoreDNS service's ClusterIP across the various CoreDNS pods.
* **High Availability:** By having multiple CoreDNS pods behind the service IP, Kubernetes ensures that if one CoreDNS pod fails, others can still handle DNS requests. This provides redundancy and high availability for DNS resolution within your cluster.
* **Simplified Configuration:** From the perspective of your microservices, they only need to be configured to use the single ClusterIP of the CoreDNS service. Kubernetes handles the complexity of routing those queries to the available DNS resolvers.

So, is it true that in advance all DNS records are stored in the CoreDNS service after the creation of the necessary services, and then when one service makes a request to another, it can easily find the appropriate DNS name and resolve it?

Yes, your understanding is largely correct! Here's a slightly more detailed explanation:

* **Service Creation and DNS Registration:** When you create a Service (like your payment-service-svc), the Kubernetes control plane automatically registers a corresponding DNS record in the CoreDNS service. This record maps the service's DNS name (e.g., payment-service-svc.default.svc.cluster.local) to its ClusterIP address (e.g., 10.0.20.20).
* **CoreDNS as the Central DNS Authority:** CoreDNS acts as the central DNS server for your Kubernetes cluster. It holds the information about all the services and their internal IP addresses.
* **On-Demand Resolution:** While the DNS records are made available to CoreDNS upon service creation, the actual resolution happens *on-demand* when a pod (like the one in order-service) makes a DNS query for another service.
* **Caching:** CoreDNS and even the operating system within the querying pod might cache the resolved IP addresses for a certain period to improve performance and reduce the load on CoreDNS for subsequent requests.

what is meant by DNS resolves to Ip address? how is it?

When we say, "DNS resolves to an IP address," it means that the Domain Name System (DNS) translates a human-friendly website name (like **google.com**) into a numerical Internet Protocol (IP) address (like **172.217.160.142**). This IP address is what computers use to identify and communicate with each other on the internet or within a network.

Think of it like a phonebook:

* **Website Name (e.g., google.com):** This is like a person's name in the phonebook – it's easy for humans to remember and use.
* **IP Address (e.g., 172.217.160.142):** This is like the phone number – it's the unique identifier that the phone system (or in this case, the internet) needs to connect you to the right person (or website).
* **DNS (Domain Name System):** This is like the phonebook itself – it's the system that holds the records of names and their corresponding numbers.

**How does this resolution happen?** Here's a simplified step-by-step process:

1. **You type a website name into your browser (e.g., google.com).**
2. **Your computer sends a DNS query to a DNS server.** This server is usually provided by your internet service provider (ISP) or can be a public DNS server (like Google's 8.8.8.8). Think of this as asking the operator for the phone number.
3. **The DNS server looks up the IP address associated with the website name.**
   * If the DNS server has this information in its cache (a temporary memory of recent lookups), it will directly return the IP address.
   * If not, it might need to query other DNS servers to find the correct IP address. This process can involve multiple servers working together, like different pages in a phonebook or different operators forwarding your request.
4. **The DNS server sends the IP address back to your computer.** This is like the operator giving you the phone number.
5. **Your computer uses the IP address to connect to the web server hosting the website.** Once your computer has the IP address, it can directly communicate with the server where the google.com website files are stored and retrieve the web page to display in your browser.

So, DNS resolution is the fundamental process that allows us to use easy-to-remember website names instead of having to memorize complex numerical IP addresses for every website we want to visit. It's a crucial part of how the internet works.

Ok, Let's take an example and explain the entire above context in real-time scenario and visualize it after that?

**Scenario: Microservices Communication in AKS**

You have an AKS cluster named **my-aks-cluster** with two microservices:

* **order-service:** Handles order processing.
* **payment-service:** Handles payment processing.

You want **order-service** to communicate with **payment-service** to process payments when a new order is placed.

**Step-by-Step Explanation:**

**1. Deployment in AKS:**

Both **order-service** and **payment-service** are deployed as pods in your AKS cluster. Initially, these pods are on the same node pool called **default-pool**.

**2. Creating ClusterIP Services:**

You create two **ClusterIP** services to enable internal communication:

* **order-service-svc:** This service exposes the order-service pods internally. Kubernetes assigns it a **ClusterIP**, say **10.0.10.10**, and a **DNS name**: order-service-svc.default.svc.cluster.local.
* **payment-service-svc:** This service exposes the payment-service pods internally. Kubernetes assigns it a **ClusterIP**, say **10.0.20.20**, and a **DNS name**: payment-service-svc.default.svc.cluster.local.

**3. order-service Needs to Communicate:**

When a user places an order, **order-service** needs to initiate a payment through **payment-service**.

**4. DNS Query (Same Node Pool):**

Inside the **order-service** code, it does **not** try to find the IP of **payment-service** pods directly. Instead, it makes a request to http://payment-service-svc. Kubernetes **automatically** appends the namespace and .svc.cluster.local suffix, so the query becomes payment-service-svc.default.svc.cluster.local.

**5. CoreDNS Resolution:**

* **order-service** pod sends a DNS query for payment-service-svc.default.svc.cluster.local to the **CoreDNS** service. The CoreDNS service in your AKS cluster has a **stable ClusterIP** address (e.g., **10.96.0.10**).
* **CoreDNS** resolves the query to **ClusterIP 10.0.20.20** of **payment-service-svc**.
* The **order-service** pod receives this resolved IP address.

**6. Traffic Routing (Same Node Pool):**

* **order-service** pod sends a request to **10.0.20.20:8080** (the IP and port of **payment-service-svc**).
* **Kubernetes kube-proxy** on the **default-pool** node intercepts this traffic and forwards it to the appropriate **payment-service** pod in the same node pool.

**Scenario Change: Different Node Pools**

Now, suppose **payment-service** pods are scheduled to run in a **different node pool** called **processing-pool**.

**7. DNS Query (Different Node Pools):**

* The process for **order-service** to query DNS remains unchanged. It still queries for **payment-service-svc.default.svc.cluster.local**.
* **CoreDNS** will resolve this query to **ClusterIP 10.0.20.20**, regardless of where the **payment-service** pods are located (even if they’re in a different node pool).

**8. Traffic Routing (Different Node Pools):**

* **order-service** pod sends traffic to **10.0.20.20** as before.
* The **kube-proxy** on the **default-pool** node where **order-service** is running will intercept the traffic. However, the **payment-service** pods are now in **processing-pool**.
* Kubernetes ensures that the traffic is routed across nodes to reach the correct pod, even if it’s on a **different node pool**.

**Text-Based Visualization:**

**Same Node Pool:**

|  |
| --- |
| +-----------------+ DNS Query: payment-service-svc.default... +-----------------+  | order-service |------------------------------------------------------->| CoreDNS (IP: 10.96.0.10) |  | (Pod in | +-----------------+  | default-pool) | |  +-----------------+ | Resolves to: 10.0.20.20  | |  | Request to 10.0.20.20:8080 v  +-----------------+----------------------------------------------------+-----------------+  | Kubernetes Node | | Kubernetes Node |  | (default-pool) | | (default-pool) |  | (kube-proxy) | | (kube-proxy) |  +-----------------+ +-----------------+  | |  | Traffic routed to pod IP of payment-service v  +-----------------+ +-----------------+  | payment-service |<--------------------------------------------| payment-service |  | (Pod in | | (Pod in |  | default-pool) | | default-pool) |  +-----------------+ +-----------------+ |

**Different Node Pools:**

|  |
| --- |
| +-----------------+ DNS Query: payment-service-svc.default... +-----------------+  | order-service |------------------------------------------------------->| CoreDNS (IP: 10.96.0.10) |  | (Pod in | +-----------------+  | default-pool) | |  +-----------------+ | Resolves to: 10.0.20.20  | |  | Request to 10.0.20.20:8080 v  +-----------------+----------------------------------------------------+-----------------+  | Kubernetes Node | | Kubernetes Node |  | (default-pool) | | (processing- |  | (kube-proxy) | | pool) |  +-----------------+ | (kube-proxy) |  | +-----------------+  | Traffic routed across nodes to pod IP of payment-service |  v v  +-----------------+ +-----------------+  | | | payment-service |  | (Potentially | | (Pod in |  | other pods) | | processing-pool)|  +-----------------+ +-----------------+ |

**Summary Highlights:**

1. **ClusterIP Services**: Provide an internal DNS name for inter-service communication within the AKS cluster.
2. **CoreDNS**: Resolves service names like payment-service-svc.default.svc.cluster.local to **ClusterIP** addresses.
3. **Traffic Routing**: Kubernetes ensures that traffic from one node pool to another is handled seamlessly by **kube-proxy** and the underlying network infrastructure, regardless of where the target pods are scheduled.

This setup allows **order-service** and **payment-service** to communicate reliably within your AKS cluster, even if they reside in different node pools.