Kubernetes Networking semantics

Per Andersson

Abstraction of K8s networking

- The manifest used to define Kubernetes entities are typically free of any sort of IP address information.
  - Service
  - Network Policy
  - Pod
  - Workload Resources
    - Deployment
    - ReplicaSet
    - StatefulSet
    - DaemonSet
    - Job
- The basic semantics of Kubernetes and the information found in the manifest defines the connectivity rules and behavior

Namespace, Name and Identity

- All entities belong to a namespace
- All entities have a name that is unique in that namespace
- All entities have a unique identifier (UID)
- The identity can be simplified to type<namespace, name>
  - namespace<name>
  - service<namespace, name>
  - networkPolicy<namespace, name>
  - pod<namespace, name>
  - deployment<namespace, name>
  - replicaSet<namespace, name>
  - daemonSet<namespace, name>
  - job<namespace, name>

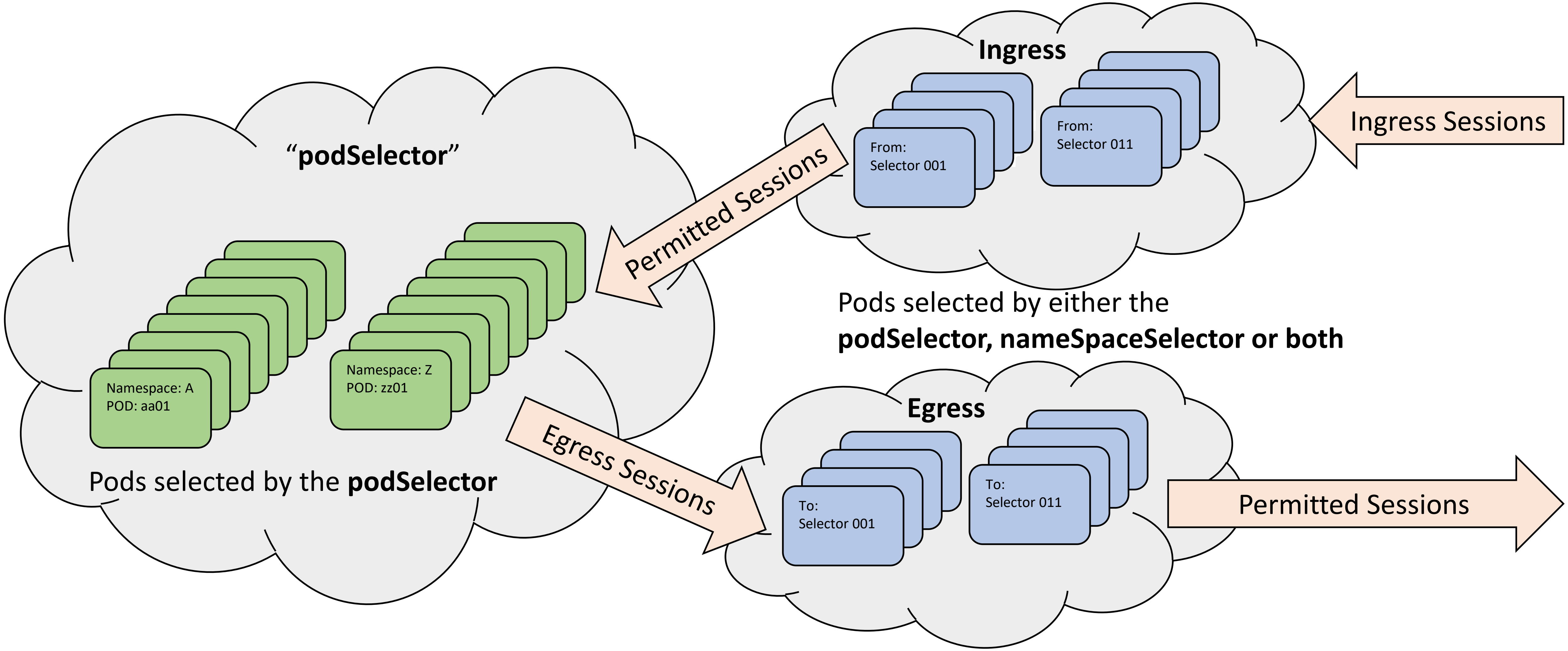
Pod lifecycle and Pod Communication

- The “Workload Resources” are there to manage and control Pods and are not basic entities from a communication standpoint, we only need to consider
  - Pod
  - Network Policy
  - Service
- The Pod life cycle can be abstracted to
  - addPod(pod<namespace, name>)
  - removePod(pod<namespace, name>)
- Kubernetes assumes that every pod can communicate with all other pod as long as there is no network policy that forbids it.
- The Pod communication can be abstracted to two connection primitives
  - openPodConnection(sourcePod, destinationPod, protocol, port) =>
    <true, connectionId> or <false, 0>
  - closePodConnection(connectionId)

Network Policy

- The NetworkPolicy spec has all the information needed to define a particular network policy in the given namespace.
  - **podSelector**: Each NetworkPolicy includes a podSelector which selects the grouping of pods to which the policy applies. An empty podSelector , ”podSelector: {} selects all pods in the namespace.
  - **policyTypes**: Each NetworkPolicy includes a policyTypes list which may include either Ingress, Egress, or both. The **policyTypes** field indicates whether or not the given policy applies to ingress traffic to selected pod, egress traffic from selected pods, or both. If no **policyTypes** are specified on a NetworkPolicy then by default Ingress will always be set and Egress will be set if the NetworkPolicy has any egress rules.
  - **ingress**: Each NetworkPolicy may include a list of whitelist ingress rules. Each rule allows traffic which matches both the **from** and **ports** sections.
  - **egress**: Each NetworkPolicy may include a list of whitelist egress rules. Each rule allows traffic which matches both the **to** and **ports** sections.
- **to** and **from** selectors, there are four kinds of selectors that can be specified in an **ingress** from section or **egress** to section:
  - **podSelector**: This selects particular Pods in the same namespace as the NetworkPolicy which should be allowed as ingress sources or egress destinations.
  - **namespaceSelector**: This selects particular namespaces for which all Pods should be allowed as ingress sources or egress destinations.
  - **namespaceSelector** *and* **podSelector**: A single **to/from** entry that specifies both **namespaceSelector** and **podSelector** selects particular Pods within particular namespaces.
  - **ipBlock**: This selects particular IP CIDR ranges to allow as ingress sources or egress destinations. These should be cluster-external IPs, since Pod IPs are ephemeral and unpredictable.

Simple Network Policy reference model



Abstract Network Policy Filter System

- addNetworkPolicy(
  networkPolicy<namespace, name)
- updateNetworkPolicy(
  networkPolicy<namespace, name)
- removeNetworkPolicy(
  networkPolicy<namespace, name)
- allowNewConnectionFromPod(
  destinationPod,
  sourcePod,
  protocol,
  destinationPort)
- allowNewConnectionToPod(
  sourcePod,
  destinationPod,
  protocol,
  destinationPort)

- The ipBlock part of to and from selectors is ignored for now
- This policy filter system is not dependent on
  - The number of pod replicas
  - The number of pod interfaces
  - The number of pod network attachments
  - Which interface an ip address is configured too
- The policy filter system is dependent on
  - Pod manifests
  - Pod labels
  - Label selectors in the Network Policies
- It is only updated when
  - policy filters are added, updated or removed
  - Labels used in policy filters are changed
- It is easy to extend with functionality
  - show how policies are related
  - which policies that applies towards a
    - namespace
    - Service
    - Workload entities
    - individual pods

Pod Communication with Network Policies

› openPodConnection must be updated to support the NW policy check

```
openPodConnection(sourcePod, destinationPod, protocol, port)
{
    if
            allowNewConnectionToPod(
                sourcePod,
                destinationPod,
                protocol,
                destinationPort)
            &&
            allowNewConnectionFromPod(
                destinationPod,
                sourcePod,
                protocol,
                destinationPort)
    then
            return <true, “NewConnectionID”>
    else
            return <false, 0>
}
```

› The allowNewConnectionToPod is used to check outgoing egress connectivity from the “sourcePod”
- This is done by matching the “sourcePod” towards the NetworkPolicy PodSelector and the “destinationPod”, “protocol” and “port” towards the NetworkPolicy egress to rules

› The allowNewConnectionToPod is used to check incomming ingress connectivity towards the “destinationPod”
- This is done by matching the “destinationPod” towards the NetworkPolicy PodSelector and the “sourcePod”, “protocol” and “port” towards the NetworkPolicy ingress from rules

Service lifecycle and Service to Pod mapping

- **Service is the basic construct used to load balance Kubernetes applications.**
  - It is built in and works in conjunction with conjunction with workload resources like Deployment, StatefulSet and DaemonSet to provide application scalability and load balancing.
  - A Service is mapped to a set of Pods by using label and label selectors, the mapping can be simplified though by defining a Service <-> Pod mapping service
- **The Service life cycle can be abstracted to**
  - addService(service<namespace, name>)
  - removeService(servcice<namespace, name>)
- **The mapping service can be abstracted to six primitives**
  - addPod2ServiceMapping(pod<namespace, name>, service<namespace, name>)
  - removePod2ServiceMapping(pod<namespace, name>, service<namespace, name>)
  - removePod2ServiceMappings(pod<namespace, name>)
  - removeService2PodMappings(service<namespace, name>)
  - lookupPod2ServiceMappings(pod<namespace, name>) => {service<namespace, name>*}
  - lookupService2PodMappings(Service<namespace, name>) => {pod<namespace, name>*}

Pod and Service lifecycle updates and service to pod communication

- Pod lifecycle: removePod must clean up the Service mappings

```
removePod(pod<namespace, name>)
{
        removePod2ServiceMappings(pod<namespace, name>)
}
```

- Service lifecycle: removeService must clean up the Pod mappings

```
removeService(service<namespace, name>)
{
        removeService2PodMappings(service<namespace, name>)
}
```

- Service to pod communication can be abstracted to one primitive
    - openServiceConnection(
        sourcePod,
        destinationService,
        protocol,
        port)
      => <true, connectionId> or <false, 0>

OpenServiceConnection can then specified as

```
openServiceConnection(sourcePod,
                      destinationService,
                      protocol,
                      port)
  {
      podSet pods =
      lookupService2PodMappings(destinationService)
      if isEmptySet(pods) then
              return <false, 0>
      else
              return openPodConnection(sourcePod,
                      pods.SelectOnePod(),
                      protocol,
                      port)
  }
```

Basic Kubernetes Network Semantics

- The basic Network semantics can be summarized in
  - All Pods can communicate with each other, unless there is a network policy rule that forbids it
  - A Service can load balance connectivity towards a set of Pods
  - The Network Policy regulate connectivity between Pods

What is missing

- Pod with single network attachment towards one network with single IP address assignment
- Pod with multi network attachment towards one or several networks with single IP Addresses assignment for each network attachment
- Pod with multi network attachment towards one or several networks with multiple IP Addresses assignment for each network attachment
- Overlapping address spaces
- Dynamic networking
  - Add/remove network
  - Pod with support for dynamic network attachment and detachment
  - Pod with support for dynamic interface address assignment and removal

Step 1:Extend model with support for single network and pods with single Network Attachment with single IP address

- Every Pod should have one IP address added to one interface that is attached to a common network
- Every Service can be assigned one Virtual IP address
- A Network Policy must support the IpBlock in egress and ingress rules
- Add support for connectivity to and from cluster external entities

Map ip addresses to Pod And Services

- Add a mapping service from Pod to ip addresses
  - addAddress2PodMapping(ipAddress, pod<namespace, name>)
  - removeAddress2PodMapping(ipAddress, pod<namespace, name>)
  - removePod2AddressMappings(pod<namespace, name>)
  - lookupPod2AddressMappings(pod<namespace, name>) => {ipAddress*}
  - lookupAddress2PodMapping(ipAddress) => {pod<namespace, name>?}
- Add a mapping service from Service to ip addresses
  - addAddress2ServiceMapping(ipaddress, service<namespace, name>)
  - removeAddress2ServiceMapping(ipaddress , service<namespace, name>)
  - removeService2AddressMappings(service<namespace, name>)
  - lookupService2AddressMappings(service<namespace, name>) => {ipAddress*}
  - lookupAddress2ServiceMapping(ipAddress=> {service<namespace, name>?}

New primitives and lifecycle updates

- Communication must be extended with two new primitives
  - openConnectionWithIpAddresses(sourceIpAddress,
    destinationIpAddress,
    protocol,
    port) => <true, connectionId> or <false, 0>
- Network Policy system must be extended with two more primitives
  - allowNewConnectionFromIpAddress(
    sourceIpAddress,
    destinationPod,
    protocol,
    destinationPort)
  - allowNewConnectionToIpAddress(
    sourcePod,
    destinationIpAddress,
    protocol,
    destinationPort)
- Pod lifecycle: removePod must clean up the Address mappings

```
removePod(pod<namespace, name>)
{
        removePod2ServiceMappings(pod<namespace, name>)
        removePod2AddressMappings(pod<namespace, name>)
}
```

- Service lifecycle: removeService must clean up the Address mappings

```
removeService(service<namespace, name>)
{
        removeService2PodMappings(service<namespace, name>)
        removeService2AddressMappings(service<namespace, name>)
}
```

Definition of openConnectionFromIpAddresses

```
openConnectionFromIpAddresses(
        sourceIpAddress,
        destinationIpAddress,
        protocol,
        port)
{
        // find out if addresses maps to pod or services
        podSet sourcePod = lookupAddress2PodMapping(sourceIpAddress)
        serviceSet destService = lookupAddress2ServiceMapping(destinationIpAddress)
        podSet destPod

        if destService.isEmpty() then {// find out if destination address maps to a pod
                destPod = lookupAddress2PodMapping(destinationIpAddress)
        } else {
                destPod = lookupService2PodMappings(destService.selectOnePod())
        }
        if destPod.isEmpty() then {
                if sourcePod.isNotEmpty() &&
                        allowNewConnectionToIpAddress(sourcePod[0],
                                destinationIpAddress,
                                protocol,
                                destinationPort) then {
                        return <true, "New ConnectionID">
                }
                return <false, 0>
        }
        if sourcePod.isEmpty() then {
                if allowNewConnectionFromIpAddress(sourceIpAddress,
                                destinationPod[0],
                                protocol,
                                destinationPort) then {
                        return <true, "New ConnectionID">
                }
        } else {
                if allowNewConnectionFromPod(sourcePod[0],
                                destinationPod[0],
                                protocol,
                                destinationPort) then {
                        return <true, "New ConnectionID">
                }
        }
        return <false, 0>
}
```

- Important to understand that everything is based on that
  - Addresses are unique
  - An address can only be used in one mapping
- From that it is possible to conclude several interesting things
  - It does not matter how many addresses that are mapped to a Pod!!!
  - The only thing that matters is that lookupAddress2PodMapping only can return zero or one Pod
  - The number of addresses returned by lookupPod2AddressMapping is irrelevant for the Kubernetes Network Semantic
- The Kubernetes Network semantics is not changed by
  - The number networks used in the system
  - The number of network attachments in a pod
  - The number of networks attached to a pod
  - The number of IP addresses assigned to an interface/network attachment
- That said, the network environment in any multi homed Pod becomes more challenging
  - How to handle routes, VRFs….

Step 2:Extend model with support for multi network and pods with multi–Network Attachments with multiple IP addresses

- Every Pod must have at least IP addresses added to an interface that is attached to a common “cluster” network
- Every Service can be assigned one Virtual IP address
- A Network Policy must support the IpBlock in egress and ingress rules
- Add support for connectivity to and from cluster external entities
- A Pod can have multiple network attachments towards one or more networks
- A Pod can have one or more ip addresses assigned to each Network attachment
- This is already supported by the model

Pod and Service Abstractions

Pod lifecycle

- addPod(pod<namespace, name>)
- removePod(pod<namespace, name>)

Service lifecycle

- addService(service<namespace, name>)
- removeService(servcice<namespace, name>)

Pod and Service Communication

- openPodConnection(sourcePod,
  destinationPod,
  protocol,
  port) =>
  <true, connectionId> or <false, 0>
- openServiceConnection(sourcePod,
  destinationService,
  protocol,
  port) =>
  <true, connectionId> or <false, 0>
- openConnectionWithIpAddresses(
  sourceIpAddress,
  destinationIpAddress,
  protocol,
  port) =>
  <true, connectionId> or <false, 0>
- closePodConnection(connectionId)

Pod to Service mapping

- addPod2ServiceMapping(pod<namespace, name>,
  service<namespace, name>)
- removePod2ServiceMapping(pod<namespace, name>,
  service<namespace, name>)
- removePod2ServiceMappings(pod<namespace, name>)
- removeService2PodMappings(service<namespace, name>)
- lookupPod2ServiceMappings(pod<namespace, name>) =>
  {service<namespace, name>*}
- lookupService2PodMappings(Service<namespace, name>) =>
  {pod<namespace, name>*}

Pod to ip addresses mapping

- addAddress2PodMapping(ipAddress, pod<namespace, name>)
- removeAddress2PodMapping(ipAddress, pod<namespace, name>)
- removePod2AddressMappings(pod<namespace, name>)
- lookupPod2AddressMappings(pod<namespace, name>) =>
  {ipAddress*}
- lookupAddress2PodMapping(ipAddress) =>
  {pod<namespace, name>?}

Service to ip address mapping

- addAddress2ServiceMapping(service<namespace, name>, ipaddress)
- removeAddress2ServiceMapping(service<namespace, name>, ipaddress)
- removeService2AddressMappings(service<namespace, name>)
- lookupService2AddressMappings(service<namespace, name>) =>
  {ipAddress*}
- lookupAddress2ServiceMapping(ipAddress) =>
  {service<namespace, name>?}

Abstract Network Policy Filter System

- Network Policy primitives
  - addNetworkPolicy(
    networkPolicy<namespace, name)
  - updateNetworkPolicy(
    networkPolicy<namespace, name)
  - removeNetworkPolicy(
    networkPolicy<namespace, name)
  - allowNewConnectionFromPod(
    destinationPod,
    sourcePod,
    protocol,
    destinationPort)
  - allowNewConnectionToPod(
    sourcePod,
    destinationPod,
    protocol,
    destinationPort)
  - allowNewConnectionFromIpAddress(
    sourceIpAddress,
    destinationPod,
    protocol,
    destinationPort)
  - allowNewConnectionToIpAddress(
    sourcePod,
    destinationIpAddress,
    protocol,
    destinationPort)

- This policy filter system is not dependent on
  - The number of pod replicas
  - The number of pod interfaces
  - The number of pod network attachments
  - Which interface an ip address is configured too
- The policy filter system is dependent on
  - Pod manifests
  - Pod labels
  - Label selectors in the Network Policies
- It is only updated when
  - policy filters are added, updated or removed
  - Labels used in policy filters are changed
- It is easy to extend with functionality
  - show how policies are related
  - which policies that applies towards a
    - namespace
    - Service
    - Workload entities
    - individual pods

Definition of primitives

- removePod

```
removePod(pod<namespace, name>)
{
        removePod2ServiceMappings(pod<namespace, name>)
        removePod2AddressMappings(pod<namespace, name>)
 }
```

- openPodConnection

```
openPodConnection(sourcePod, destinationPod, protocol, port)
 {
        if
                 allowNewConnectionToPod(
                        sourcePod,
                        destinationPod,
                        protocol,
                        destinationPort)
                &&
                 allowNewConnectionFromPod(
                        destinationPod,
                        sourcePod,
                        protocol,
                        destinationPort)
        then
                return <true, “NewConnectionID”>
        else
                return <false, 0>
}
```

- removeService

```
removeService(service<namespace, name>)
{
        removeService2PodMappings(service<namespace, name>)
        removeService2AddressMappings(service<namespace,
name>)
 }
```

OpenServiceConnection

```
openServiceConnection(sourcePod,
                destinationService,
                protocol,
                port)
    {
        podSet pods =
        lookupService2PodMappings(destinationService)
        if isEmptySet(pods) then {
                return <false, 0>
        } else {
                return openPodConnection(sourcePod,
                        pods.SelectOnePod(),
                        protocol,
                        port)
        }
    }
```

openConnectionFromIpAddresses

```
openConnectionFromIpAddresses(
        sourceIpAddress,
        destinationIpAddress,
        protocol,
        port)
{
        // find out if addresses maps to pod or services
        podSet sourcePod = lookupAddress2PodMapping(sourceIpAddress)
        serviceSet destService = lookupAddress2ServiceMapping(destinationIpAddress)
        podSet destPod

        if destService.isEmpty() then {// find out if destination address maps to a pod
                destPod = lookupAddress2PodMapping(destinationIpAddress)
        } else {
                destPod = lookupService2PodMappings(destService.selectOnePod())
        }
        if destPod.isEmpty() then {
                if sourcePod.isNotEmpty() &&
                        allowNewConnectionToIpAddress(sourcePod[0],
                                destinationIpAddress,
                                protocol,
                                destinationPort) then {
                        return <true, “New ConnectionID”>
                }
                return <false, 0>
        }
        if sourcePod.isEmpty() then {
                if allowNewConnectionFromIpAddress(sourceIpAddress,
                                destinationPod[0],
                                protocol,
                                destinationPort) then {
                        return <true, “New ConnectionID”>
                }
        } else {
                if openPodConnection(sourcePod[0],
                                destinationPod[0],
                                protocol,
                                destinationPort) then {
                        return <true, “New ConnectionID”>
                }
        }
        return <false, 0>
}
```