Sirius API

# Overview

This document covers the Sirius API proposal for configuring of the SDN Control Plane on dedicated hardware and requirements related to implementation.

# Scope

The following is in scope of the document:

* Creation/Deletion of the Interface
* ACLs
* Routing
* Mapping
* Load Balancing
* Transforms
* Billing
* Telemetry
* Diagnostics

# Definitions

|  |  |
| --- | --- |
| Name | Definition |
| ACL | Access Control List |
| ARP | Address Resolution Protocol |
| ASN | Autonomous System Number |
| AZ | Availability Zone |
| Baremetal | Entire node not running any custom SDN software (connected to “SDN”/”VNET” via “intelligent” router/appliance) |
| BGP | Border Gateway Protocol |
| CA | Customer Address (visible inside Customer VM) |
| Cloud | Top Level (collection of Regions) |
| Cluster | Collection of Nodes across different Racks |
| Connection | Control plane communication between sender and receiver (usually involves handshake) |
| DC | Data Center |
| DHCP | Dynamic Host Configuration Protocol (IPv4) |
| DHCPv6 | Dynamic Host Configuration Protocol (IPv6) |
| DSR | Direct Server Return |
| DSR VIP | Virtual IP used for Direct Server Return |
| ELB | External Load Balancer |
| ENI | Elastic Network Interface |
| ER | Express Route (CISCO/ARISTA/etc hardware that performs point-to-point connectivity and forwards traffic) |
| Flow | Single transposition (TBD – expand) – Data plane stream of packets between sender and receiver that shares key IP header information |
| Flow Table | Collection of Flows |
| gRPC | Google RPC |
| GW | Gateway |
| HA | High Availability |
| ILB | Internal Load Balancer |
| ILB | Internal Load Balancer |
| IPSec | IPSec Tunnel or IPSec Device |
| IPv4 | IP protocol Version 4 (ex. 10.1.2.3) |
| IPv6 | IP protocol Version 6 (ex. 2001:1234:abcd::1) |
| JSON | JavaScript Script Object Notation |
| LB | Load Balancer |
| LPM | Longest-Prefix-Match algorithm commonly used in routing. |
| MAC | MAC Address (Media Access Control) |
| Mapping | Mapping transformation between CA:PA:MAC |
| NA | Neighbor Advertisement |
| NDP | NDP |
| Node / Blade | Single Physical Machine in a Rack |
| NS | Neighbor Solicitation |
| NVA | Network Virtual Appliance (VM that might have forwarding or filtering functionality – ex. router or firewall deployed as Linux/Windows VM/baremetal appliance). |
| NVGRE / GRE | Generic Routing Encapsulation (Protocol) |
| Overlay |  |
| PA | Provider Address (internal Azure Datacenter address used for routing) |
| Peering | Network Relationship between two entities (usually between two VNETs – ex. VNET Peering) |
| PhyNet | Physical Network |
| Prefix | For IPv4: (0-32) – example 10.0.0.0/8  For IPv6: (0-128) – example: 2001:1234:abcd::/48 |
| RA | Router Advertisement |
| Rack | Collection of physical machines, switches etc |
| Region | Single Region (collection of DCs in same region) |
| RS | Router Solicitation |
| SDN | Software Defined Networking (high level name for the Virtual Network and its elements) |
| SONIC | SONIC Switch Platform |
| Spoof Guard | Rule put in place to prevent VM from spoofing traffic |
| TCP | Transmission Control Protocol |
| TOR | Top of the Rack Switch (aka ToR or T0) |
| UDP | User Datagram Protocol |
| Underlay |  |
| VFP | Virtual Filtering Platform |
| VIP | Virtual IP (IP exposed on Load Balancer) |
| VM | Virtual Machine |
| VNET | Virtual Network |
| VXLAN | VXLAN Encapsulation Protocol |
| XML | Extensible Markup Language (Format) |

# Assumptions

Processing Pipeline must support both IPv4 and IPv6 protocols for both Underlay and Overlay (unless explicitly stated that some scenarios is IPv4-only or IPv6-only).

# Processing Pipeline

## ENI selection

Once packet arrives on Inbound to the device/card, it must be forwarded to the correct ENI policy processing pipeline.

This ENI selection will be done based on the Inner Destination MAC of the packet, which will be matched against the MAC of the ENI.

## Policy processing per ENI

Packets will be processed via the following pipeline.

### Outbound

VM->ACLs->Routing->Network

### Inbound

Network->Routing->ACLs->VM

## Important considerations

Packets coming from “Network” might be of the following types:

1. Encapped within VNET traffic (from VM to VM)
2. Encapped traffic from MUX to VM
3. Encapped traffic from Device to VM
4. Direct traffic from infrastructure to VM (ex. Node to VM) (no encap)

Packet going outside to “Network” might be of the following types:

1. Direct traffic to Internet (no encap)
2. Direct traffic to infrastructure (no encap)
3. Encapped within VNET traffic (from VM to VM)
4. Encapped traffic from VM to Device

# API Definition

## Creation/Deletion of Interface

Each Interface, also called Elastic Network Interface (ENI), is independent entity that has collection of routing policies.

Usually there is 1:1 mapping between VM (or Physical) NIC and the ENI (Virtual NIC).

VM/Physical NIC represents the NIC exposed directly to the VM.

### CreateEni

This API creates an elastic network interface with specified ENI match criterion.

First version of this API only support mac-address as ENI identification criterion.

ENI identification criterion is also used to identify packet direction.

For ENI created with identifier Mac1, it assumes packets with destination mac as Mac1 are inbound and packets with source mac as Mac2 are outbound. This direction is used for matching appropriate inbound and outbound policies.

Usage:

|  |
| --- |
| message Guid {  bytes Bytes = 1; // A 16-element byte array  }  message MacAddress {  bytes Bytes = 1; // A 6-element byte array  }  message EniId {  Guid eni\_id = 1;  }  rpc AddEni(EniId, MacAddress) returns(); |

### DeleteEni

This API removes the elastic network interface with specified ENI identifier and any policy associated with this ENI.

Usage:

|  |
| --- |
| message EniId {  Guid eni\_id = 1;  }  rpc DeleteEni(EniId) returns(); |

## Access Control List

ACLs must support multiple level/groups ACLing. Packet successfully must pass thru the groups in order to be moved to the Routing layer.

We need to support up to 3 (possibly 5?) ACL groups in each direction. Order of ACL groups evaluation will always be the same and will not change.

Example:

**Outbound**

Order of evaluation if there if no flow: VM->ACLStage1->ACLStage2->ACLStage3->Routing

**Inbound**

Order of evaluation if no flow: Routing->ACLStage1-> ACLStage2->ACLStage3-> VM

Each ACL group will be having distinct set of rules.

ACLs will be evaluated in both Inbound and Outbound direction. There will be separate ACL groups for Inbound and separate ACL groups for Outbound.

Updating of the ACL Group (ACLStage) must be an **atomic** operation. **No partial updates is allowed**, as it might lead to security issues in case only partial rules are applied.

**Rules Evaluation logic**

ACL logic will at the end evaluate packet into single outcome: ALLOW or DENY.

If ALLOW outcome is reached – packet will be moved to next processing pipeline.

If DENY outcome is reached – packet will be **silently** dropped.

ACL groups needs to be evaluated in order.

Each ACL group will have set of rules. **Only single rule can match in group/stage**. Once the rule is matched, its action is performed (ALLOW/DENY) and we move to next ACL group/stage (once we find the match, no further rules in same group are being evaluated).

Within ACL Group rules will be organized by priority (with lowest priority number being evaluated first). No two rules will have same priority within a group. Priority is only withing rules in same group. No priorities across groups (no mix).

**Terminating vs non-Terminating behavior**

Rule can be “terminating” or “non-terminating”. Terminating rule means this is the **final outcome** and further processing thru other groups/stages must be skipped. DENY rules are usually “terminating”.

ALLOW rules are usually “non-terminating”, as we need to have packet allowed by all groups.

DENY rule can sometimes be also “non-terminating” (soft DENY). This means that particular ACL group “proposes” to DENY the packet, but its decision is not final, and can be “overridden” (switched) to ALLOW by the next group/stage.

ACL rules (within each group separately) are organized by priority. Smaller priority number means rule will be evaluated first.

Priorities are unique withing an ACL group. Priorities might overlap across ACL groups.

Priority is only indicating priority of a rules within a group for the purpose of evaluating this group rules only. Priorities does not spawn across different ACL groups.

The following processing algorithm is proposed:

|  |
| --- |
| set **OUTCOME** to DENY (default value)  foreach ACL group {  find first rule that matches (based on order of increasing priority value)  once rule gets matched, set OUTCOME based on that rule  if rule is terminating, finish entire ACL pipeline processing and move to next pipeline  if rule is non-terminating, finish the current group and move to next ACL group/stage  }  if all ACL groups/stage were processed, ALLOW/DENY packet based on the OUTCOME |

ACLs must be stateful.

Example of what it means to be stateful:

* Customer has ALLOW for Outbound Traffic (to Internet), but DENY for Inbound (from Internet)
* VM must be able to “initiate” traffic outbound (which will be allowed by outbound ALLOW rule). This should then automatically create “temporary” inbound allow rule for that specific flow ONLY to allow Internet machine to reply back (with SYN-ACK).
* Internet must not be able to initiate connection to VM if there is DENY Inbound rule.

### SetACL

ACL evaluation is done in stages, where stage1 ACL are evaluated first, if packet is allowed through stage1 it gets processed by stage2 ACL and so on. For packet to be allowed it must be allowed in all 3 stages or must hit a terminating allow rule.

|  |
| --- |
| message Stage {  uint8 stage\_index = 1;  }  enum Direction {  INBOUND = 0;  OUTBOUND = 1;  }  message AclGroup {  repeated Acl acl = 1;  }  message Acl {  string name = 1;  uint16 priority = 2;  AclAction action = 3;  repeated uint8 protocols = 4;  repeated PortRange source\_port\_ranges = 5;  repeated IPSubnet source\_address\_prefixes = 6;  repeated PortRange destination\_port\_ranges = 7;  repeated IPSubnet destination\_address\_prefixes = 8;  bool terminating = 9;  }  message IPAddress {  bytes Bytes = 1; // A 4-element byte array for IPv4 address, or 16-element byte array for IPv6 address.  }  message IPSubnet {  IPAddress network\_address = 1;  uint8 prefix\_length = 2;  }  message PortRange {  uint16 StartPort = 1;  uint16 EndPort = 2;  }  enum AclAction {  DENY = 0;  ALLOW = 1;  }  rpc SetAcl(EniId, Stage, Direction, AclGroup) returns(); |

### GetACLs

Get the group and all the ACLs.

|  |
| --- |
| rpc GetAcl(EniId, Stage, Direction) returns(AclGroup); |

### Example ACL rules

|  |  |  |
| --- | --- | --- |
| Rules | Packet | Outcome |
|  |  |  |
|  |  |  |

## Routing

Routing must be LPM based.

Routing must support all combinations of underlay/overlay:

* inner IPv4 packet encapped in outer IPv4 packet
* inner IPv4 packet encapped in outer IPv6 packet
* inner IPv6 packet encapped in outer IPv4 packet
* inner IPv6 packet encapped in outer IPv6 packet

Routing pipeline must support the following routing models.

Outbound

1. Transpositions
   1. Direct traffic – pass thru with static SNAT/DNAT (IP, IP+Port)
   2. Packet upcasting (IPv4 -> IPv6 packet transformation)
   3. Packet downcasting (IPv6 -> IPv4 packet transformation)
2. Encap
   1. VXLAN/GRE encap – static rule
   2. VXLAN/GRE encap – based on mapping lookup
   3. VXLAN/GRE encap – calculated based on part of SRC/DEST IP of inner packet
3. Up to 3 level of routing transforms (example: transpose + encap + encap)

Potential representation in hardware tables:

|  |
| --- |
| ----------------------------------------------------------------  | PREFIX | TYPE1 | TBL1IDX | TYPE2 | TBL2IDX | TYPE3 | TBL3IDX |  ---------------------------------------------------------------- |

Inbound

1. Decap
   1. VXLAN/GRE decap – static rule
   2. VXLAN/GRE decap – based on mapping lookup
   3. VXLAN/GRE decap – inner packet SRC/DEST IP calculated based on part of outer packet SRC/DEST IP
2. Transpositions
   1. Direct traffic – pass thru with static SNAT/DNAT (IP, IP+Port)
   2. Packet upcasting (IPv4 -> IPv6 packet transformation)
   3. Packet downcasting (IPv6 -> IPv4 packet transformation)
3. Up to 3 level of routing transforms (example: decap + decap + transpose)

All routing rules must optionally allow for “stamping” source MAC (to **enforce Source MAC correctness**). Correct/fix/override source mac.

### Routes rules processing

#### Outbound (LPM)

Matching is based on Destination IP only - using the **Longest-Prefix-Match (LPM)** algorithm.

Once the rule is match, the correct set of transposition, encap steps must be applied depending on the rule.

Only one rules will be matched.

#### Inbound (Priority)

All inbound rules are matched based on the **priority order** (with lower priority value rule matched first).

Matching is based on multiple fields (or must match if field is populated). Supported fields:

* Most Outer Source IP Prefix
* Most Outer Destination IP Prefix
* VXLAN/GRE key

Once the rule is match, the correct set of decap, transposition steps must be applied depending on the rule.

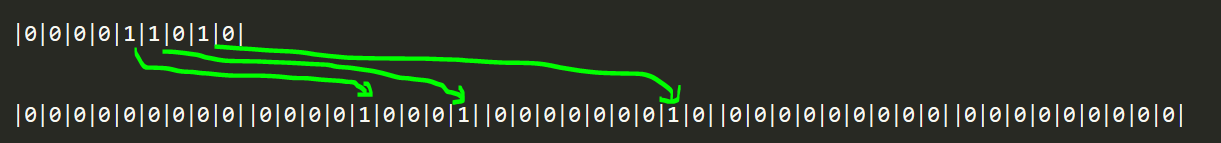
Only one rules will be matched.

### SetRoute

SddRoute API adds routes for outbound packet processing.

VNET, NAT, Private Link, ST are all modelled as route action, depending on LPM appropriate route action is executed.

|  |
| --- |
| message Routes {  oneof routes {  RoutesOutbound outbound = 1;  RoutesInbound inbound = 2;  }  }  message RoutesOutbound {  repeated Route route = 1;  }  message RoutesInbound {  repeated RouteInbound route = 1;  }  message RouteInbound {  uint16 priority = 1;  Vni vni = 2; // match against vni, 0 == not using for matching  Protocol procotol = 3; // 0 (any), 6 (TCP), 17 (UDP)  IPSubnet source\_prefix = 4; // 0.0.0.0/0 == not using for matching  uint16 source\_port = 5; // 0 == not using for matching  IPSubnet destination\_prefix = 6; // 0.0.0.0/0 == not using for matching  uint16 destination\_port = 7; // 0 == not using for matching  repeated RoutingTransformInbound routing\_transform = 8;  optional MeteringBucket metering\_bucket = 9;  }  message RoutingTransformInbound {  oneof transform {  Transform transposition = 1;  DecapStatic static\_encap = 2;  DecapMapping mapping\_encap = 3;  DecapCalculation calculation\_encap = 4;  }  }  message DecapStatic {  }  message DecapMapping {  }  message BitMask {  repeated bytes = 1;  }  message DecapCalculation {  IPSubnet source\_ip\_prefix = 1;  IPSubnet destination\_ip\_prefix = 2;  BitMask source\_ip\_from\_mask = 3;  BitMask source\_ip\_to\_mask = 4;  BitMask destination\_ip\_from\_mask = 5;  BitMask destination\_ip\_to\_mask = 6;  }  message RouteOutbound {  repeated IPSubnet destination\_prefixes = 1;  repeated RoutingTransformOutbound routing\_transform = 2; // up to 3 (5?) routing transformations  optional MeteringBucket metering\_bucket = 3;  }  message RoutingTransformOutbound {  oneof transform {  Transform transform = 1;  TransformCalculation calculation\_transform = 2;  TransformMapping mapping\_transform = 3;  EncapStatic static\_encap = 4;  EncapMapping mapping\_encap = 5;  EncapCalculation calculation\_encap = 6;  }  }  message TransformMapping {    }  message Transform {  IPAddress source\_ip = 1; // 0.0.0.0 == no modification of IP  uint16 source\_port = 2; // port 0 == no modification of port  MacAddress source\_mac = 3; // 00:00:00:00:00:00 == no modification of MAC  IPAddress destination\_ip = 4; // 0.0.0.0 == no modification of IP  uint16 destination\_port = 5; // port 0 == no modification of port  MacAddress destination\_mac = 6; // 00:00:00:00:00:00 == no modification of MAC  }  message TransformCalculation {  IPSubnet source\_ip\_prefix = 1;  BitMask source\_to\_source\_from\_mask = 2;  BitMask source\_to\_source\_to\_mask = 3;  IPSubnet destination\_ip\_prefix = 4;  BitMask dest\_to\_dest\_from\_mask = 5;  BitMask dest\_to\_dest\_to\_mask = 6;  BitMask source\_to\_dest\_from\_mask = 7;  BitMask source\_to\_dest\_to\_mask = 8;  BitMask dest\_to\_source\_from\_mask = 9;  BitMask dest\_to\_source\_to\_mask = 10;  }  message NextHop {  IPAddress destination\_ip = 1;  Vni vni = 2;  }  message EncapStatic {  EncapType type = 1;  repeated NextHop next\_hop = 2; // ECMP  }  message EncapMapping {  EncapType type = 1;  }  message BitMask {  repeated bytes = 1;  }  message EncapCalculation {  EncapType type = 1;  IPSubnet source\_ip\_prefix = 2;  BitMask source\_to\_source\_from\_mask = 3;  BitMask source\_to\_source\_to\_mask = 4;  IPSubnet destination\_ip\_prefix = 5;  BitMask dest\_to\_dest\_from\_mask = 6;  BitMask dest\_to\_dest\_to\_mask = 7;  BitMask source\_to\_dest\_from\_mask = 8;  BitMask source\_to\_dest\_to\_mask = 9;  BitMask dest\_to\_source\_from\_mask = 10;  BitMask dest\_to\_source\_to\_mask = 11;  Vni vni = 12;  }  enum EncapType {  VXLAN = 0;  NVGRE = 1;  GENEVE = 2;  }  rpc AddRouteOutbound(EniId, Routes) returns();  rpc AddRouteInbound(EniId, Routes) returns(); |



|  |  |
| --- | --- |
| Route Type | Example |
| Encap\_with\_lookup | * Encap action is executed based on lookup into the **mapping table**. * V4 underlay is used. * Destination mac is re-written with mac in mapping. * Encap is decided based on route metadata – VXLAN, Geneve etc. * Undelay is decided based on route metadata. * Depending on mapping type appropriate transform is done for PL / VNET. |
| Encap\_with\_Provided\_data | * Encap action is executed based on provided data in routing metadate. |
| Outbound NAT (SNAT)\_L3 | * L3 NAT action is executed on source IP, based on provided data. |
| Outbound NAT (SNAT)\_L4 | * L4 NAT action is executed on source IP, source port based on provided data. |
| Null | * Blocks the traffic. |
| Service\_Tunnel | * Do ST transform based on specified route parameter |

|  |  |
| --- | --- |
| Mapping Type | Example |
| VNET\_Mapping | 10.0.0.1 – Pav4 – Pav6 – MAC1 – VNET\_Mapping |
| Private Link Mapping | 20.0.0.1 – PA1 – MAC1 – PL\_Mapping – Redirect map Id |

### Scenarios

#### VNET Traffic

|  |
| --- |
| Mapping {  { 10.0.0.5, Map { 100.0.0.1, ::0, AB:CD:EF:12:34:05, 7771 }, 0 },  { 10.0.0.6, Map { 100.0.0.2, ::0, AB:CD:EF:12:34:06, 7771 }, 0 }  }  // VNET  // 10.0.0.5 -> 10.0.0.6 ==> [100.0.0.1][100.0.0.2][7771]10.0.0.5|10.0.0.6  RouteOutbound {  { 10/8 },  EncapMapping { IPv4 },  1 // metering bucket for in-VNET traffic  }  // VNET  // [100.0.0.2][100.0.0.1][7771]10.0.0.6|10.0.0.5 ===> 10.0.0.6 -> 10.0.0.5  RouteInbound {  15,  7771, // VNI  0, // any protocol  0.0.0.0, 0,  100.0.0.1, 0,  DecapMapping { IPv4 },  1 // metering bucket for in-VNET traffic  } |

#### Regional Peering

|  |
| --- |
| Mapping {  { 10.0.0.5, Map { 100.0.0.1, ::0, AB:CD:EF:12:34:05, 7771 }, 0 },  { 10.1.0.6, Map { 100.0.0.2, ::0, AB:CD:EF:12:34:06, 7771 }, 0 }  }  // VNET - Regional Peering  // 10.0.0.5 -> 10.1.0.6 ==> [100.0.0.1][100.0.0.2][7771]10.0.0.5|10.1.0.6  RouteOutbound {  { 10/8 },  EncapMapping { IPv4 },  2 // metering bucket for regionally-peered-VNET traffic  }  // VNET - Regional Peering  // [100.0.0.2][100.0.0.1][7772]10.0.0.6|10.0.0.5 ===> 10.1.0.6 -> 10.0.0.5  RouteInbound {  15,  7772, // VNI  0, // any protocol  0.0.0.0, 0,  100.0.0.1, 0,  DecapMapping { IPv4 },  2 // metering bucket for regionally-peered-VNET traffic  } |

#### Global Peering

|  |
| --- |
| Mapping {  { 10.0.0.5, Map { 100.0.0.1, 2603:10e1::1, AB:CD:EF:12:34:05, 7771 }, 0 },  { 11.0.0.6, Map { 100.0.0.2, 2603:10e1::2, AB:CD:EF:12:34:06, 7773 }, 0 },  }  // VNET - Global Peering  // 10.0.0.5 -> 11.0.0.6 ==> [2603:10e1::1][2603:10e1::2][7771]10.0.0.5|11.0.0.6  RouteOutbound {  { 10/8 },  EncapMapping { IPv6 },  3 // metering bucket for globally-peered-VNET traffic  }  // VNET - Global Peering  // [2603:10e1::1][2603:10e1::2][7773]11.0.0.6|10.0.0.5 ===> 11.0.0.6 -> 10.0.0.5  RouteInbound {  15,  7772, // VNI  0, // any protocol  0.0.0.0, 0,  2603:10e1::2, 0,  DecapMapping { IPv6 },  3 // metering bucket for globally-peered-VNET traffic  } |

#### Outbound SNAT (via ManagedNAT)

|  |
| --- |
| // Outbound SNAT (via ManagedNAT)  // 10.0.0.5 -> 8.8.8.8 ===> 51.52.53.54[10.0.0.5] -> 8.8.8.8  RouteOutbound {  { 0/0 },  EncapStatic { NVGRE, NextHop { 51.52.53.54, 200 } },  5555 // special metering bucket (metering group)  } |

#### ILPIP

|  |
| --- |
| // Inbound DNAT (from Internet - ILPIP)  // 8.8.8.8 -> 21.22.23.24 (VIP) ===> 8.8.8.8 -> 10.0.0.5 (CA)  RouteInbound {  50,  0, // VNI  0, // any protocol  0.0.0.0, 0,  21.22.23.24, 0,  Transform { 0.0.0.0, 0, 00:00:00:00:00:00,  10.0.0.5, 0, 00:00:00:00:00:00 },  5555 // special metering bucket (metering group)  }  // Outbound SNAT (to Internet - ILPIP)  // 10.0.0.5 (CA) -> 8.8.8.8 ===> 21.22.23.24 (VIP) -> 8.8.8.8  RouteOutbound {  { 0/0 },  Transform { 21.22.23.24, 0, 00:00:00:00:00:00,  0.0.0.0, 0, 00:00:00:00:00:00 },  5555 // special metering bucket (metering group)  } |

#### VIP

|  |
| --- |
| // Inbound VIP (from Internet)  // 8.8.8.8 -> 21.22.23.24:80 (VIP:PORT) -> 8.8.8.8 ===> 8.8.8.8 -> 10.0.0.5:8080 (CA:PORT)  RouteInbound {  50,  0, // VNI  0, // any protocol  0.0.0.0, 0,  21.22.23.24, 80,  Transform { 0.0.0.0, 0, 00:00:00:00:00:00,  10.0.0.5, 8080, 00:00:00:00:00:00 },  5555 // special metering bucket (metering group)  }  // Outbound VIP (to Internet)  // 10.0.0.5:8080 (CA:PORT) -> 8.8.8.8 ===> 21.22.23.24:80 (VIP:PORT) -> 8.8.8.8  RouteOutbound {  { 0/0 },  Transform { 21.22.23.24, 80, 00:00:00:00:00:00,  0.0.0.0, 0, 00:00:00:00:00:00 },  5555 // special metering bucket (metering group)  } |

#### Service Tunnel

|  |
| --- |
| ----------------------------------------------------------------  | PREFIX | TYPE1 | TBL1IDX | TYPE2 | TBL2IDX | TYPE3 | TBL3IDX |  ----------------------------------------------------------------  // Service Tunnel  // 10.0.0.5 -> 51.1.2.3 ===> [IPv4][IPv4]fd00:3401:203:d204:0:200:a00:5 -> 2603:10e1:100:2::3401:203  RouteOutbound {  { 52.1.2.3/32, 21.0.0.0/8 },  TransformCalculation { fd00:3401:203:d204:0:200::0, "FFFF:FFFF", "0:0:0:0:0:0:FFFF:FFFF",  2603:10e1:100:2::0, "FFFF:FFFF", "0:0:0:0:0:0:FFFF:FFFF",  0, 0,  "FFFF:FFFF", "0:FFFF:FFFF:0:0:0:0:0" }, // dest-to-source  EncapCalculation { NVGRE, 20.21.22.23, 0, 0,  0.0.0.0, "0:0:0:0:0:0:FFFF:FFFF", "FFFF:FFFF",  0, 0,  0, 0, 100},  12345 // metering bucket  }  // Service Tunnel  // 2603:10e1:100:2::3401:203 -> fd00:3401:203:d204:0:200:a00:5 ===> 51.1.2.3 -> 10.0.0.5  RouteInbound {  10,  100, // VNI  0, // any protocol  0.0.0.0, 0,  0.0.0.0, 0,  DecapStatic,  TransformCalculation { 0.0.0.0, "0:0:0:0:0:0:FFFF:FFFF", "FFFF:FFFF",  0.0.0.0, "0:0:0:0:0:0:FFFF:FFFF", "FFFF:FFFF",  0, 0,  0, 0 },  12345 // metering bucket  } |

#### Private Link

|  |
| --- |
| ------------------------  | IP | TYPE1 | TBL1IDX |  ------------------------  ---------------------------------------  | IDX | PA IPv4 | PA IPv6 | MAC | GRE |  ---------------------------------------  -------------------------------------------------------------  | IDX | TYPE1 | TBL1IDX | TYPE2 | TBL2IDX | TYPE3 | TBL3IDX |  -------------------------------------------------------------  Mapping {  { 10.0.0.5, Map { 100.0.0.1, 2603:10e1::1, AB:CD:EF:12:34:05, 7771 }, 0 },  { 10.0.0.6, Map { 100.0.0.2, 2603:10e1::2, AB:CD:EF:12:34:06, 7773 }, 0 },  { 10.0.0.11, Route {  TransformCalculation { fd40:108:0:d204:0:200::0, "FFFF:FFFF", "0:0:0:0:0:0:FFFF:FFFF",  2603:10e1:100:2::3401:203, 0, 0  0, 0,  "FFFF:FFFF", "0:FFFF:FFFF:0:0:0:0:0" }, // dest-to-source  EncapCalculation { NVGRE, 52.1.2.3, 0, 0,  0.0.0.0, "0:0:0:0:0:0:FFFF:FFFF", "FFFF:FFFF",  0, 0,  0, 0, 100 },  },  1234 // metering bucket per private link  }  }  // Private Link  // 10.0.0.5 -> 10.0.0.11[LinkID=2049,VIP=52.1.2.3] ===> fd00:108:0:d204:0:200:a00:5 -> 2603:10e1:100:2::3401:203  RouteOutbound {  { 10/8 },  EncapMapping { Route },  0 // metering bucket not used (mapping bucket will be used)  }  // Private Link  // 2603:10e1:100:2::3401:203 -> fd00:108:0:d204:0:200:a00:5 ===> 10.0.0.11 -> 10.0.0.5  RouteInbound {  10,  100, // VNI  0, // any protocol  0.0.0.0, 0,  0.0.0.0, 0,  DecapStatic,  TransformCalculation { 0.0.0.0, "0:0:0:0:0:0:FFFF:FFFF", "FFFF:FFFF",  0.0.0.0, "0:0:0:0:0:0:FFFF:FFFF", "FFFF:FFFF",  0, 0,  0, 0 },  12345 // metering bucket  } |

## Mapping management

### SetMapping

Enables adding overlay-underlay mapping information.

|  |
| --- |
| message Mappings {  repeated Mapping = 1;  }  message Mapping {  IPAddress overlay\_ip = 1;  oneof mapping {  Map map = 2;  Route route = 3;  }  MeteringBucketId bucket\_id = 4;  }  message Map {  IPAddress underlay\_ipv4 = 2;  IPAddress underlay\_ipv6 = 3;  MacAddress mac = 4;  Vni vni = 5;  }  message MeteringBucketId {  uint32 id = 1;  }  rpc SetMappings(EniId, Mappings) returns(); |

### RemoveMapping

|  |
| --- |
| rpc RemoveMappings(EniId, Mappings) returns(); |

### GetMappings

|  |
| --- |
| rpc GetMappings(EniId) returns(Mappings); |

### ClearMappings

|  |
| --- |
| rpc ClearMappings(EniId) returns(); |

## Billing and Metering

Packets flowing thru hardware pipeline must be counted to allow for billing customers based on traffic.

Categories of what is differently billed:

* In-VNET traffic (traffic between VMs in same VNET)
* Cross-VNET traffic (traffic between VM in VNET A and VM in VNET B)
* Traffic to infrastructure devices (ex. ER device, node)
* Traffic to Internet (any externally/internally routable VIP) inside same Azure AZ
* Traffic to Internet (any externally/internally routable VIP) inside same Azure Region
* Traffic to Internet (any externally/internally routable VIP) to/from different Azure Region
* Traffic to Internet (any externally/internally routable VIP) outside of Azure

There will be multiple distinct counters for each of the above category based on:

* Outbound: destination routing rule + destination IP
* Inbound: source routing rule + source IP

Customer is billed based on number of bytes sent/received separately. Distinct counter must be supported for outbound vs inbound traffic of each category.

Counters will be created by SDN Control Plane.

**Important consideration**

Billing must not be based on Destination IP, but on routing rule that is being hit.

Routing rules are controlled by customer and customer might have different confiugrations:

1. 0/0 -> Route to Internet (bill as Internet Traffic)
2. 0/0 -> Route to NVA which is a VM deployed in Peered VNET (bill as Peering Traffic going to the specific peered VNET)

Different routing rules will be generated depending on how customer decides to configure routing (different a vs b). Each rule will have counter associated to inform processing pipeline which counter is to be incremented.

**Size considerations**

All buckets must be UINT64 size and start from value 0.

**What is being counted**

Buckets will be counting number of bytes (of the RAW/Original packet).

* Outbound – count bytes of the packet that VM sent before any routing transforms were applied (before packet is encapped)
* Inbound – count bytes of the packet after all the routing transforms were applied (after packet is decapped, transformed, etc)

### AddMeteringBucket

Add new Metering Bucket of specific ID.

|  |
| --- |
| message MeteringBucketId {  uint32 id = 1;  }  rpc AddMeteringBucket(EniId, MeteringBucketId) returns(); |

### RemoveMeteringBucket

Removes the Metering Bucket of specific ID.

|  |
| --- |
| rpc RemoveMeteringBucket(EniId, MeteringBucketId) returns(); |

This call must fail if there is any existing rule that is associated with that metering bucket ID (if this metering bucket is still in use).

### GetMeteringBucket

Retrieves metering backet current value based on specific bucket ID.

|  |
| --- |
| rpc GetMeteringBucket(EniId, MeteringBucketId) returns(uint64); |

### ResetMeteringBucket

Reset metering bucket value back to 0.

|  |
| --- |
| rpc ResetMeteringBucket(EniId, MeteringBucketId) returns(); |

### SetMeteringGroup

|  |
| --- |
| message MeteringGroup {  repeated MeteringRule rule = 1;  }  message MeteringRule {  uint16 priority = 1;  repeated IPSubnet destination\_prefixes = 1;  MeteringBucketId bucket\_id = 2;  }  rpc SetMeteringGroup(EniId, MeteringBucketId, Direction, MeteringGroup) returns(); |

## Port State Management

Used to Unblock and Block port.

### SetPortState

|  |
| --- |
| message PortState {  bool enabled = 1;  }  rpc SetPortState(EniId, PortState) returns(); |

### GetPortState

|  |
| --- |
| rpc GetPortState(EniId) returns(PortState); |

## Flow Management

### GetFlows

TBD

### ClearFlows

TBD

# Examples

## Sample Port config for VNET-VNET

|  |
| --- |
| VNET – 10.0.0.0/24, 20.0.0.0/24 , VNET GRE = 50000  SRC = 10.0.0.1 , DST = 20.0.0.1  ENIDId eniId = AddENI(InnerMac, “08-D2-3E-68-89-C7”)  AddACL(eniId, 0, List<> SubnetACLRules) // Subnet NSG  AddACL(eniId, 1, List<> NICACLRules) // NIC NSG  AddRoute(eniId, {10.0.0.0/24, Encap\_with\_lookup})  AddRoute(eniId, {20.0.0.0/24, Encap\_with\_lookup})  AddMapping(eniId, {10.0.0.1, PA1v4, PA1V6, MAC1, VXLAN } )  AddMapping(eniId, {20.0.0.1, PA2v4, PA2V6, MAC2, VXLAN } ) |