NAME

ovn-architecture − Open Virtual Network architecture

DESCRIPTION

OVN, the Open Virtual Network, is a system to support virtual network abstraction. OVN complements the existing capabilities of OVS to add native support for virtual network abstractions, such as virtual L2 and L3 overlays and security groups. Services such as DHCP are also desirable features. Just like OVS, OVN's design goal is to have a production-quality implementation that can operate at significant scale.

An OVN deployment consists of several components:

- A *Cloud Management System* (*CMS*), which is OVN's ultimate client (via its users and administrators). OVN integration requires installing a CMS-specific plugin and related software (see below). OVN initially targets OpenStack as CMS.

  We generally speak of ''the'' CMS, but one can imagine scenarios in which multiple CMSes manage different parts of an OVN deployment.

- An OVN Database physical or virtual node (or, eventually, cluster) installed in a central location.

- One or more (usually many) *hypervisors*. Hypervisors must run Open vSwitch and implement the interface described in **IntegrationGuide.rst** in the OVS source tree. Any hypervisor platform supported by Open vSwitch is acceptable.

- Zero or more *gateways*. A gateway extends a tunnel-based logical network into a physical network by bidirectionally forwarding packets between tunnels and a physical Ethernet port. This allows non-virtualized machines to participate in logical networks. A gateway may be a physical host, a virtual machine, or an ASIC-based hardware switch that supports the **vtep**(5) schema.

  Hypervisors and gateways are together called *transport node* or *chassis*.

The diagram below shows how the major components of OVN and related software interact. Starting at the top of the diagram, we have:

- The Cloud Management System, as defined above.

- The *OVN/CMS Plugin* is the component of the CMS that interfaces to OVN. In OpenStack, this is a Neutron plugin. The plugin's main purpose is to translate the CMS's notion of logical network configuration, stored in the CMS's configuration database in a CMS-specific format, into an intermediate representation understood by OVN.

  This component is necessarily CMS-specific, so a new plugin needs to be developed for each CMS that is integrated with OVN. All of the components below this one in the diagram are CMS-independent.

- The *OVN Northbound Database* receives the intermediate representation of logical network configuration passed down by the OVN/CMS Plugin. The database schema is meant to be ''impedance matched'' with the concepts used in a CMS, so that it directly supports notions of logical switches, routers, ACLs, and so on. See **ovn−nb**(5) for details.

  The OVN Northbound Database has only two clients: the OVN/CMS Plugin above it and **ovn−northd** below it.

- **ovn−northd**(8) connects to the OVN Northbound Database above it and the OVN Southbound Database below it. It translates the logical network configuration in terms of conventional network concepts, taken from the OVN Northbound Database, into logical datapath flows in the OVN Southbound Database below it.

- The *OVN Southbound Database* is the center of the system. Its clients are **ovn−northd**(8) above it and **ovn−controller**(8) on every transport node below it.

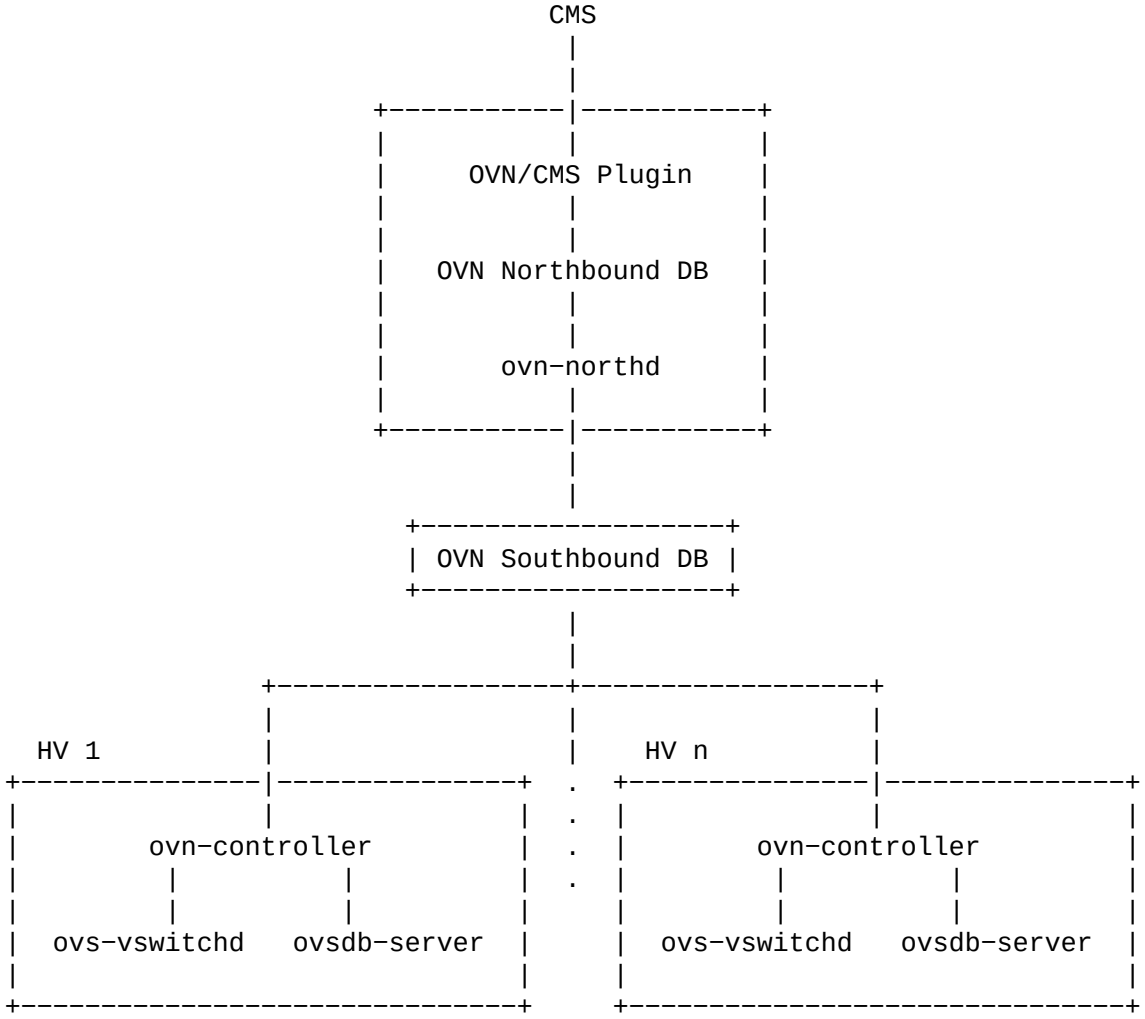
  The OVN Southbound Database contains three kinds of data: *Physical Network* (PN) tables that specify how to reach hypervisor and other nodes, *Logical Network* (LN) tables

that describe the logical network in terms of ‘‘logical datapath flows,’’ and *Binding* tables that link logical network components’ locations to the physical network. The hypervisors populate the PN and Port_Binding tables, whereas **ovn−northd**(8) populates the LN tables.

OVN Southbound Database performance must scale with the number of transport nodes. This will likely require some work on **ovsdb−server**(1) as we encounter bottlenecks. Clustering for availability may be needed.

The remaining components are replicated onto each hypervisor:

- **ovn−controller**(8) is OVN’s agent on each hypervisor and software gateway. Northbound, it connects to the OVN Southbound Database to learn about OVN configuration and status and to populate the PN table and the **Chassis** column in **Binding** table with the hypervisor’s status. Southbound, it connects to **ovs−vswitchd**(8) as an OpenFlow controller, for control over network traffic, and to the local **ovsdb−server**(1) to allow it to monitor and control Open vSwitch configuration.
- **ovs−vswitchd**(8) and **ovsdb−server**(1) are conventional components of Open vSwitch.



Information Flow in OVN

Configuration data in OVN flows from north to south. The CMS, through its OVN/CMS plugin, passes the logical network configuration to **ovn−northd** via the northbound database. In turn, **ovn−northd** compiles the configuration into a lower-level form and passes it to all of the chassis via the southbound database.

Status information in OVN flows from south to north. OVN currently provides only a few forms of status information. First, **ovn−northd** populates the **up** column in the northbound **Logical_Switch_Port** table: if a logical port's **chassis** column in the southbound **Port_Binding** table is nonempty, it sets **up** to **true**, otherwise to **false**. This allows the CMS to detect when a VM's networking has come up.

Second, OVN provides feedback to the CMS on the realization of its configuration, that is, whether the configuration provided by the CMS has taken effect. This feature requires the CMS to participate in a sequence number protocol, which works the following way:

1. When the CMS updates the configuration in the northbound database, as part of the same transaction, it increments the value of the **nb_cfg** column in the **NB_Global** table. (This is only necessary if the CMS wants to know when the configuration has been realized.)
2. When **ovn−northd** updates the southbound database based on a given snapshot of the northbound database, it copies **nb_cfg** from northbound **NB_Global** into the southbound database **SB_Global** table, as part of the same transaction. (Thus, an observer monitoring both databases can determine when the southbound database is caught up with the northbound.)
3. After **ovn−northd** receives confirmation from the southbound database server that its changes have committed, it updates **sb_cfg** in the northbound **NB_Global** table to the **nb_cfg** version that was pushed down. (Thus, the CMS or another observer can determine when the southbound database is caught up without a connection to the southbound database.)
4. The **ovn−controller** process on each chassis receives the updated southbound database, with the updated **nb_cfg**. This process in turn updates the physical flows installed in the chassis's Open vSwitch instances. When it receives confirmation from Open vSwitch that the physical flows have been updated, it updates **nb_cfg** in its own **Chassis** record in the southbound database.
5. **ovn−northd** monitors the **nb_cfg** column in all of the **Chassis** records in the southbound database. It keeps track of the minimum value among all the records and copies it into the **hv_cfg** column in the northbound **NB_Global** table. (Thus, the CMS or another observer can determine when all of the hypervisors have caught up to the northbound configuration.)

Chassis Setup

Each chassis in an OVN deployment must be configured with an Open vSwitch bridge dedicated for OVN's use, called the *integration bridge*. System startup scripts may create this bridge prior to starting **ovn−controller** if desired. If this bridge does not exist when ovn-controller starts, it will be created automatically with the default configuration suggested below. The ports on the integration bridge include:

- On any chassis, tunnel ports that OVN uses to maintain logical network connectivity. **ovn−controller** adds, updates, and removes these tunnel ports.
- On a hypervisor, any VIFs that are to be attached to logical networks. The hypervisor itself, or the integration between Open vSwitch and the hypervisor (described in **IntegrationGuide.rst**) takes care of this. (This is not part of OVN or new to OVN; this is preexisting integration work that has already been done on hypervisors that support OVS.)
- On a gateway, the physical port used for logical network connectivity. System startup scripts add this port to the bridge prior to starting **ovn−controller**. This can be a patch port to another bridge, instead of a physical port, in more sophisticated setups.

Other ports should not be attached to the integration bridge. In particular, physical ports attached to the underlay network (as opposed to gateway ports, which are physical ports attached to logical networks) must not be attached to the integration bridge. Underlay physical ports should instead be attached to a separate Open vSwitch bridge (they need not be attached to any bridge at all, in fact).

The integration bridge should be configured as described below. The effect of each of these settings is documented in **ovs−vswitchd.conf.db**(5):

fail−mode=secure
Avoids switching packets between isolated logical networks before **ovn−controller** starts up. See **Controller Failure Settings** in **ovs−vsctl**(8) for more information.

other−config:disable−in−band=true
Suppresses in-band control flows for the integration bridge. It would be unusual for such flows to show up anyway, because OVN uses a local controller (over a Unix domain socket) instead of a remote controller. It's possible, however, for some other bridge in the same system to have an in-band remote controller, and in that case this suppresses the flows that in-band control would ordinarily set up. Refer to the documentation for more information.

The customary name for the integration bridge is **br−int**, but another name may be used.

Logical Networks

A *logical network* implements the same concepts as physical networks, but they are insulated from the physical network with tunnels or other encapsulations. This allows logical networks to have separate IP and other address spaces that overlap, without conflicting, with those used for physical networks. Logical network topologies can be arranged without regard for the topologies of the physical networks on which they run.

Logical network concepts in OVN include:

- *Logical switches*, the logical version of Ethernet switches.
- *Logical routers*, the logical version of IP routers. Logical switches and routers can be connected into sophisticated topologies.
- *Logical datapaths* are the logical version of an OpenFlow switch. Logical switches and routers are both implemented as logical datapaths.
- *Logical ports* represent the points of connectivity in and out of logical switches and logical routers. Some common types of logical ports are:
  - Logical ports representing VIFs.
  - *Localnet ports* represent the points of connectivity between logical switches and the physical network. They are implemented as OVS patch ports between the integration bridge and the separate Open vSwitch bridge that underlay physical ports attach to.
  - *Logical patch ports* represent the points of connectivity between logical switches and logical routers, and in some cases between peer logical routers. There is a pair of logical patch ports at each such point of connectivity, one on each side.
  - *Localport ports* represent the points of local connectivity between logical switches and VIFs. These ports are present in every chassis (not bound to any particular one) and traffic from them will never go through a tunnel. A **localport** is expected to only generate traffic destined for a local destination, typically in response to a request it received. One use case is how OpenStack Neutron uses a **localport** port for serving metadata to VM's residing on every hypervisor. A metadata proxy process is attached to this port on every host and all VM's within the same network will reach it at the same IP/MAC address without any traffic being sent over a tunnel. Further details can be seen at https://docs.openstack.org/developer/networking-ovn/design/metadata_api.html.

Life Cycle of a VIF

Tables and their schemas presented in isolation are difficult to understand. Here's an example.

A VIF on a hypervisor is a virtual network interface attached either to a VM or a container running directly on that hypervisor (This is different from the interface of a container running inside a VM).

The steps in this example refer often to details of the OVN and OVN Northbound database schemas. Please

see **ovn−sb**(5) and **ovn−nb**(5), respectively, for the full story on these databases.

1. A VIF's life cycle begins when a CMS administrator creates a new VIF using the CMS user interface or API and adds it to a switch (one implemented by OVN as a logical switch). The CMS updates its own configuration. This includes associating unique, persistent identifier *vif-id* and Ethernet address *mac* with the VIF.

2. The CMS plugin updates the OVN Northbound database to include the new VIF, by adding a row to the **Logical_Switch_Port** table. In the new row, **name** is *vif-id*, **mac** is *mac*, **switch** points to the OVN logical switch's Logical_Switch record, and other columns are initialized appropriately.

3. **ovn−northd** receives the OVN Northbound database update. In turn, it makes the corresponding updates to the OVN Southbound database, by adding rows to the OVN Southbound database **Logical_Flow** table to reflect the new port, e.g. add a flow to recognize that packets destined to the new port's MAC address should be delivered to it, and update the flow that delivers broadcast and multicast packets to include the new port. It also creates a record in the **Binding** table and populates all its columns except the column that identifies the **chassis**.

4. On every hypervisor, **ovn−controller** receives the **Logical_Flow** table updates that **ovn−northd** made in the previous step. As long as the VM that owns the VIF is powered off, **ovn−controller** cannot do much; it cannot, for example, arrange to send packets to or receive packets from the VIF, because the VIF does not actually exist anywhere.

5. Eventually, a user powers on the VM that owns the VIF. On the hypervisor where the VM is powered on, the integration between the hypervisor and Open vSwitch (described in **IntegrationGuide.rst**) adds the VIF to the OVN integration bridge and stores *vif-id* in **external_ids**:**iface−id** to indicate that the interface is an instantiation of the new VIF. (None of this code is new in OVN; this is pre-existing integration work that has already been done on hypervisors that support OVS.)

6. On the hypervisor where the VM is powered on, **ovn−controller** notices **external_ids**:**iface−id** in the new Interface. In response, in the OVN Southbound DB, it updates the **Binding** table's **chassis** column for the row that links the logical port from **external_ids**: **iface−id** to the hypervisor. Afterward, **ovn−controller** updates the local hypervisor's OpenFlow tables so that packets to and from the VIF are properly handled.

7. Some CMS systems, including OpenStack, fully start a VM only when its networking is ready. To support this, **ovn−northd** notices the **chassis** column updated for the row in **Binding** table and pushes this upward by updating the **up** column in the OVN Northbound database's **Logical_Switch_Port** table to indicate that the VIF is now up. The CMS, if it uses this feature, can then react by allowing the VM's execution to proceed.

8. On every hypervisor but the one where the VIF resides, **ovn−controller** notices the completely populated row in the **Binding** table. This provides **ovn−controller** the physical location of the logical port, so each instance updates the OpenFlow tables of its switch (based on logical datapath flows in the OVN DB **Logical_Flow** table) so that packets to and from the VIF can be properly handled via tunnels.

9. Eventually, a user powers off the VM that owns the VIF. On the hypervisor where the VM was powered off, the VIF is deleted from the OVN integration bridge.

10. On the hypervisor where the VM was powered off, **ovn−controller** notices that the VIF was deleted. In response, it removes the **Chassis** column content in the **Binding** table for the logical port.

11. On every hypervisor, **ovn−controller** notices the empty **Chassis** column in the **Binding** table's row for the logical port. This means that **ovn−controller** no longer knows the physical location of the logical port, so each instance updates its OpenFlow table to reflect that.

12. Eventually, when the VIF (or its entire VM) is no longer needed by anyone, an administrator deletes the VIF using the CMS user interface or API. The CMS updates its own configuration.

13. The CMS plugin removes the VIF from the OVN Northbound database, by deleting its row in the **Logical_Switch_Port** table.

14. **ovn−northd** receives the OVN Northbound update and in turn updates the OVN Southbound database accordingly, by removing or updating the rows from the OVN Southbound database **Logical_Flow** table and **Binding** table that were related to the now-destroyed VIF.

15. On every hypervisor, **ovn−controller** receives the **Logical_Flow** table updates that **ovn−northd** made in the previous step. **ovn−controller** updates OpenFlow tables to reflect the update, although there may not be much to do, since the VIF had already become unreachable when it was removed from the **Binding** table in a previous step.

Life Cycle of a Container Interface Inside a VM

OVN provides virtual network abstractions by converting information written in OVN_NB database to OpenFlow flows in each hypervisor. Secure virtual networking for multi-tenants can only be provided if OVN controller is the only entity that can modify flows in Open vSwitch. When the Open vSwitch integration bridge resides in the hypervisor, it is a fair assumption to make that tenant workloads running inside VMs cannot make any changes to Open vSwitch flows.

If the infrastructure provider trusts the applications inside the containers not to break out and modify the Open vSwitch flows, then containers can be run in hypervisors. This is also the case when containers are run inside the VMs and Open vSwitch integration bridge with flows added by OVN controller resides in the same VM. For both the above cases, the workflow is the same as explained with an example in the previous section ("Life Cycle of a VIF").

This section talks about the life cycle of a container interface (CIF) when containers are created in the VMs and the Open vSwitch integration bridge resides inside the hypervisor. In this case, even if a container application breaks out, other tenants are not affected because the containers running inside the VMs cannot modify the flows in the Open vSwitch integration bridge.

When multiple containers are created inside a VM, there are multiple CIFs associated with them. The network traffic associated with these CIFs need to reach the Open vSwitch integration bridge running in the hypervisor for OVN to support virtual network abstractions. OVN should also be able to distinguish network traffic coming from different CIFs. There are two ways to distinguish network traffic of CIFs.

One way is to provide one VIF for every CIF (1:1 model). This means that there could be a lot of network devices in the hypervisor. This would slow down OVS because of all the additional CPU cycles needed for the management of all the VIFs. It would also mean that the entity creating the containers in a VM should also be able to create the corresponding VIFs in the hypervisor.

The second way is to provide a single VIF for all the CIFs (1:many model). OVN could then distinguish network traffic coming from different CIFs via a tag written in every packet. OVN uses this mechanism and uses VLAN as the tagging mechanism.

1. A CIF's life cycle begins when a container is spawned inside a VM by the either the same CMS that created the VM or a tenant that owns that VM or even a container Orchestration System that is different than the CMS that initially created the VM. Whoever the entity is, it will need to know the *vif-id* that is associated with the network interface of the VM through which the container interface's network traffic is expected to go through. The entity that creates the container interface will also need to choose an unused VLAN inside that VM.

2. The container spawning entity (either directly or through the CMS that manages the underlying infrastructure) updates the OVN Northbound database to include the new CIF, by adding a row to the **Logical_Switch_Port** table. In the new row, **name** is any unique identifier, **parent_name** is the *vif-id* of the VM through which the CIF's network traffic is expected to go through and the **tag** is the VLAN tag that identifies the network traffic of that CIF.

3. **ovn−northd** receives the OVN Northbound database update. In turn, it makes the corresponding updates to the OVN Southbound database, by adding rows to the OVN Southbound database's **Logical_Flow** table to reflect the new port and also by creating a new row in the **Binding** table and populating all its columns except the column that identifies the **chassis**.

4. On every hypervisor, **ovn−controller** subscribes to the changes in the **Binding** table. When a new row is created by **ovn−northd** that includes a value in **parent_port** column of **Binding** table, the **ovn−controller** in the hypervisor whose OVN integration bridge has that same value in *vif-id* in **external_ids**:**iface−id** updates the local hypervisor's OpenFlow tables so that packets to and from the VIF with the particular VLAN **tag** are properly handled. Afterward it updates the **chassis** column of the **Binding** to reflect the physical location.

5. One can only start the application inside the container after the underlying network is ready. To support this, **ovn−northd** notices the updated **chassis** column in **Binding** table and updates the **up** column in the OVN Northbound database's **Logical_Switch_Port** table to indicate that the CIF is now up. The entity responsible to start the container application queries this value and starts the application.

6. Eventually the entity that created and started the container, stops it. The entity, through the CMS (or directly) deletes its row in the **Logical_Switch_Port** table.

7. **ovn−northd** receives the OVN Northbound update and in turn updates the OVN Southbound database accordingly, by removing or updating the rows from the OVN Southbound database **Logical_Flow** table that were related to the now-destroyed CIF. It also deletes the row in the **Binding** table for that CIF.

8. On every hypervisor, **ovn−controller** receives the **Logical_Flow** table updates that **ovn−northd** made in the previous step. **ovn−controller** updates OpenFlow tables to reflect the update.

Architectural Physical Life Cycle of a Packet

This section describes how a packet travels from one virtual machine or container to another through OVN. This description focuses on the physical treatment of a packet; for a description of the logical life cycle of a packet, please refer to the **Logical_Flow** table in **ovn−sb**(5).

This section mentions several data and metadata fields, for clarity summarized here:

tunnel key
: When OVN encapsulates a packet in Geneve or another tunnel, it attaches extra data to it to allow the receiving OVN instance to process it correctly. This takes different forms depending on the particular encapsulation, but in each case we refer to it here as the ''tunnel key.'' See **Tunnel Encapsulations**, below, for details.

logical datapath field
: A field that denotes the logical datapath through which a packet is being processed. OVN uses the field that OpenFlow 1.1+ simply (and confusingly) calls ''metadata'' to store the logical datapath. (This field is passed across tunnels as part of the tunnel key.)

logical input port field
: A field that denotes the logical port from which the packet entered the logical datapath. OVN stores this in Open vSwitch extension register number 14.

  Geneve and STT tunnels pass this field as part of the tunnel key. Although VXLAN tunnels do not explicitly carry a logical input port, OVN only uses VXLAN to communicate with gateways that from OVN's perspective consist of only a single logical port, so that OVN can set the logical input port field to this one on ingress to the OVN logical pipeline.

logical output port field
: A field that denotes the logical port from which the packet will leave the logical datapath. This is initialized to 0 at the beginning of the logical ingress pipeline. OVN stores this in Open vSwitch extension register number 15.

  Geneve and STT tunnels pass this field as part of the tunnel key. VXLAN tunnels do not transmit the logical output port field. Since VXLAN tunnels do not carry a logical output port field in the tunnel key, when a packet is received from VXLAN tunnel by an OVN hypervisor, the packet is resubmitted to table 8 to determine the output port(s); when the

packet reaches table 32, these packets are resubmitted to table 33 for local delivery by checking a MLF_RCV_FROM_VXLAN flag, which is set when the packet arrives from a VXLAN tunnel.

conntrack zone field for logical ports
: A field that denotes the connection tracking zone for logical ports. The value only has local significance and is not meaningful between chassis. This is initialized to 0 at the beginning of the logical ingress pipeline. OVN stores this in Open vSwitch extension register number 13.

conntrack zone fields for routers
: Fields that denote the connection tracking zones for routers. These values only have local significance and are not meaningful between chassis. OVN stores the zone information for DNATting in Open vSwitch extension register number 11 and zone information for SNATing in Open vSwitch extension register number 12.

logical flow flags
: The logical flags are intended to handle keeping context between tables in order to decide which rules in subsequent tables are matched. These values only have local significance and are not meaningful between chassis. OVN stores the logical flags in Open vSwitch extension register number 10.

VLAN ID
: The VLAN ID is used as an interface between OVN and containers nested inside a VM (see **Life Cycle of a container interface inside a VM**, above, for more information).

Initially, a VM or container on the ingress hypervisor sends a packet on a port attached to the OVN integration bridge. Then:

1. OpenFlow table 0 performs physical-to-logical translation. It matches the packet's ingress port. Its actions annotate the packet with logical metadata, by setting the logical datapath field to identify the logical datapath that the packet is traversing and the logical input port field to identify the ingress port. Then it resubmits to table 8 to enter the logical ingress pipeline.

   Packets that originate from a container nested within a VM are treated in a slightly different way. The originating container can be distinguished based on the VIF-specific VLAN ID, so the physical-to-logical translation flows additionally match on VLAN ID and the actions strip the VLAN header. Following this step, OVN treats packets from containers just like any other packets.

   Table 0 also processes packets that arrive from other chassis. It distinguishes them from other packets by ingress port, which is a tunnel. As with packets just entering the OVN pipeline, the actions annotate these packets with logical datapath and logical ingress port metadata. In addition, the actions set the logical output port field, which is available because in OVN tunneling occurs after the logical output port is known. These three pieces of information are obtained from the tunnel encapsulation metadata (see **Tunnel Encapsulations** for encoding details). Then the actions resubmit to table 33 to enter the logical egress pipeline.

2. OpenFlow tables 8 through 31 execute the logical ingress pipeline from the **Logical_Flow** table in the OVN Southbound database. These tables are expressed entirely in terms of logical concepts like logical ports and logical datapaths. A big part of **ovn−controller**'s job is to translate them into equivalent OpenFlow (in particular it translates the table numbers: **Logical_Flow** tables 0 through 23 become OpenFlow tables 8 through 31).

   Each logical flow maps to one or more OpenFlow flows. An actual packet ordinarily matches only one of these, although in some cases it can match more than one of these flows (which is not a problem because all of them have the same actions). **ovn−controller** uses the first 32 bits of the logical flow's UUID as the cookie for its OpenFlow flow or flows. (This is not necessarily unique, since the first 32 bits of a logical flow's UUID is not necessarily unique.)

Some logical flows can map to the Open vSwitch "conjunctive match" extension (see **ovs-fields**(7)). Flows with a **conjunction** action use an OpenFlow cookie of 0, because they can correspond to multiple logical flows. The OpenFlow flow for a conjunctive match includes a match on **conj_id**.

Some logical flows may not be represented in the OpenFlow tables on a given hypervisor, if they could not be used on that hypervisor. For example, if no VIF in a logical switch resides on a given hypervisor, and the logical switch is not otherwise reachable on that hypervisor (e.g. over a series of hops through logical switches and routers starting from a VIF on the hypervisor), then the logical flow may not be represented there.

Most OVN actions have fairly obvious implementations in OpenFlow (with OVS extensions), e.g. **next;** is implemented as **resubmit**, *field* **=** *constant***;** as **set_field**. A few are worth describing in more detail:

output:

Implemented by resubmitting the packet to table 32. If the pipeline executes more than one **output** action, then each one is separately resubmitted to table 32. This can be used to send multiple copies of the packet to multiple ports. (If the packet was not modified between the **output** actions, and some of the copies are destined to the same hypervisor, then using a logical multicast output port would save bandwidth between hypervisors.)

get_arp(*P***,** *A***);**
get_nd(*P***,** *A***);**

Implemented by storing arguments into OpenFlow fields, then resubmitting to table 66, which **ovn-controller** populates with flows generated from the **MAC_Binding** table in the OVN Southbound database. If there is a match in table 66, then its actions store the bound MAC in the Ethernet destination address field.

(The OpenFlow actions save and restore the OpenFlow fields used for the arguments, so that the OVN actions do not have to be aware of this temporary use.)

put_arp(*P***,** *A***,** *E***);**
put_nd(*P***,** *A***,** *E***);**

Implemented by storing the arguments into OpenFlow fields, then outputting a packet to **ovn-controller**, which updates the **MAC_Binding** table.

(The OpenFlow actions save and restore the OpenFlow fields used for the arguments, so that the OVN actions do not have to be aware of this temporary use.)

3. OpenFlow tables 32 through 47 implement the **output** action in the logical ingress pipeline. Specifically, table 32 handles packets to remote hypervisors, table 33 handles packets to the local hypervisor, and table 34 checks whether packets whose logical ingress and egress port are the same should be discarded.

   Logical patch ports are a special case. Logical patch ports do not have a physical location and effectively reside on every hypervisor. Thus, flow table 33, for output to ports on the local hypervisor, naturally implements output to unicast logical patch ports too. However, applying the same logic to a logical patch port that is part of a logical multicast group yields packet duplication, because each hypervisor that contains a logical port in the multicast group will also output the packet to the logical patch port. Thus, multicast groups implement output to logical patch ports in table 32.

   Each flow in table 32 matches on a logical output port for unicast or multicast logical ports that include a logical port on a remote hypervisor. Each flow's actions implement sending a packet to the port it matches. For unicast logical output ports on remote hypervisors, the actions set the tunnel key to the correct value, then send the packet on the tunnel port to the correct hypervisor. (When the remote hypervisor receives the packet, table 0 there will recognize it as a tunneled packet and pass it along to table 33.) For multicast logical output ports,

the actions send one copy of the packet to each remote hypervisor, in the same way as for unicast destinations. If a multicast group includes a logical port or ports on the local hypervisor, then its actions also resubmit to table 33. Table 32 also includes:

- A higher-priority rule to match packets received from VXLAN tunnels, based on flag MLF_RCV_FROM_VXLAN, and resubmit these packets to table 33 for local delivery. Packets received from VXLAN tunnels reach here because of a lack of logical output port field in the tunnel key and thus these packets needed to be submitted to table 8 to determine the output port.
- A higher-priority rule to match packets received from ports of type **localport**, based on the logical input port, and resubmit these packets to table 33 for local delivery. Ports of type **localport** exist on every hypervisor and by definition their traffic should never go out through a tunnel.
- A fallback flow that resubmits to table 33 if there is no other match.

Flows in table 33 resemble those in table 32 but for logical ports that reside locally rather than remotely. For unicast logical output ports on the local hypervisor, the actions just resubmit to table 34. For multicast output ports that include one or more logical ports on the local hypervisor, for each such logical port *P*, the actions change the logical output port to *P*, then resubmit to table 34.

A special case is that when a localnet port exists on the datapath, remote port is connected by switching to the localnet port. In this case, instead of adding a flow in table 32 to reach the remote port, a flow is added in table 33 to switch the logical outport to the localnet port, and resubmit to table 33 as if it were unicasted to a logical port on the local hypervisor.

Table 34 matches and drops packets for which the logical input and output ports are the same and the MLF_ALLOW_LOOPBACK flag is not set. It resubmits other packets to table 40.

4. OpenFlow tables 40 through 63 execute the logical egress pipeline from the **Logical_Flow** table in the OVN Southbound database. The egress pipeline can perform a final stage of validation before packet delivery. Eventually, it may execute an **output** action, which **ovn−controller** implements by resubmitting to table 64. A packet for which the pipeline never executes **output** is effectively dropped (although it may have been transmitted through a tunnel across a physical network).

   The egress pipeline cannot change the logical output port or cause further tunneling.

5. Table 64 bypasses OpenFlow loopback when MLF_ALLOW_LOOPBACK is set. Logical loopback was handled in table 34, but OpenFlow by default also prevents loopback to the OpenFlow ingress port. Thus, when MLF_ALLOW_LOOPBACK is set, OpenFlow table 64 saves the OpenFlow ingress port, sets it to zero, resubmits to table 65 for logical-to-physical transformation, and then restores the OpenFlow ingress port, effectively disabling OpenFlow loopback prevents. When MLF_ALLOW_LOOPBACK is unset, table 64 flow simply resubmits to table 65.

6. OpenFlow table 65 performs logical-to-physical translation, the opposite of table 0. It matches the packet's logical egress port. Its actions output the packet to the port attached to the OVN integration bridge that represents that logical port. If the logical egress port is a container nested with a VM, then before sending the packet the actions push on a VLAN header with an appropriate VLAN ID.

Logical Routers and Logical Patch Ports

Typically logical routers and logical patch ports do not have a physical location and effectively reside on every hypervisor. This is the case for logical patch ports between logical routers and logical switches behind those logical routers, to which VMs (and VIFs) attach.

Consider a packet sent from one virtual machine or container to another VM or container that resides on a different subnet. The packet will traverse tables 0 to 65 as described in the previous section **Architectural Physical Life Cycle of a Packet**, using the logical datapath representing the logical switch that the sender

is attached to. At table 32, the packet will use the fallback flow that resubmits locally to table 33 on the same hypervisor. In this case, all of the processing from table 0 to table 65 occurs on the hypervisor where the sender resides.

When the packet reaches table 65, the logical egress port is a logical patch port. The implementation in table 65 differs depending on the OVS version, although the observed behavior is meant to be the same:

- In OVS versions 2.6 and earlier, table 65 outputs to an OVS patch port that represents the logical patch port. The packet re-enters the OpenFlow flow table from the OVS patch port's peer in table 0, which identifies the logical datapath and logical input port based on the OVS patch port's OpenFlow port number.
- In OVS versions 2.7 and later, the packet is cloned and resubmitted directly to the first OpenFlow flow table in the ingress pipeline, setting the logical ingress port to the peer logical patch port, and using the peer logical patch port's logical datapath (that represents the logical router).

The packet re-enters the ingress pipeline in order to traverse tables 8 to 65 again, this time using the logical datapath representing the logical router. The processing continues as described in the previous section **Architectural Physical Life Cycle of a Packet**. When the packet reachs table 65, the logical egress port will once again be a logical patch port. In the same manner as described above, this logical patch port will cause the packet to be resubmitted to OpenFlow tables 8 to 65, this time using the logical datapath representing the logical switch that the destination VM or container is attached to.

The packet traverses tables 8 to 65 a third and final time. If the destination VM or container resides on a remote hypervisor, then table 32 will send the packet on a tunnel port from the sender's hypervisor to the remote hypervisor. Finally table 65 will output the packet directly to the destination VM or container.

The following sections describe two exceptions, where logical routers and/or logical patch ports are associated with a physical location.

Gateway Routers

A *gateway router* is a logical router that is bound to a physical location. This includes all of the logical patch ports of the logical router, as well as all of the peer logical patch ports on logical switches. In the OVN Southbound database, the **Port_Binding** entries for these logical patch ports use the type **l3gateway** rather than **patch**, in order to distinguish that these logical patch ports are bound to a chassis.

When a hypervisor processes a packet on a logical datapath representing a logical switch, and the logical egress port is a **l3gateway** port representing connectivity to a gateway router, the packet will match a flow in table 32 that sends the packet on a tunnel port to the chassis where the gateway router resides. This processing in table 32 is done in the same manner as for VIFs.

Gateway routers are typically used in between distributed logical routers and physical networks. The distributed logical router and the logical switches behind it, to which VMs and containers attach, effectively reside on each hypervisor. The distributed router and the gateway router are connected by another logical switch, sometimes referred to as a **join** logical switch. On the other side, the gateway router connects to another logical switch that has a localnet port connecting to the physical network.

When using gateway routers, DNAT and SNAT rules are associated with the gateway router, which provides a central location that can handle one-to-many SNAT (aka IP masquerading).

Distributed Gateway Ports

Distributed gateway ports are logical router patch ports that directly connect distributed logical routers to logical switches with localnet ports.

The primary design goal of distributed gateway ports is to allow as much traffic as possible to be handled locally on the hypervisor where a VM or container resides. Whenever possible, packets from the VM or container to the outside world should be processed completely on that VM's or container's hypervisor, eventually traversing a localnet port instance on that hypervisor to the physical network. Whenever possible, packets from the outside world to a VM or container should be directed through the physical network directly to the VM's or container's hypervisor, where the packet will enter the integration bridge through a

localnet port.

In order to allow for the distributed processing of packets described in the paragraph above, distributed gateway ports need to be logical patch ports that effectively reside on every hypervisor, rather than **l3gateway** ports that are bound to a particular chassis. However, the flows associated with distributed gateway ports often need to be associated with physical locations, for the following reasons:

- The physical network that the localnet port is attached to typically uses L2 learning. Any Ethernet address used over the distributed gateway port must be restricted to a single physical location so that upstream L2 learning is not confused. Traffic sent out the distributed gateway port towards the localnet port with a specific Ethernet address must be sent out one specific instance of the distributed gateway port on one specific chassis. Traffic received from the localnet port (or from a VIF on the same logical switch as the localnet port) with a specific Ethernet address must be directed to the logical switch's patch port instance on that specific chassis.

  Due to the implications of L2 learning, the Ethernet address and IP address of the distributed gateway port need to be restricted to a single physical location. For this reason, the user must specify one chassis associated with the distributed gateway port. Note that traffic traversing the distributed gateway port using other Ethernet addresses and IP addresses (e.g. one-to-one NAT) is not restricted to this chassis.

  Replies to ARP and ND requests must be restricted to a single physical location, where the Ethernet address in the reply resides. This includes ARP and ND replies for the IP address of the distributed gateway port, which are restricted to the chassis that the user associated with the distributed gateway port.

- In order to support one-to-many SNAT (aka IP masquerading), where multiple logical IP addresses spread across multiple chassis are mapped to a single external IP address, it will be necessary to handle some of the logical router processing on a specific chassis in a centralized manner. Since the SNAT external IP address is typically the distributed gateway port IP address, and for simplicity, the same chassis associated with the distributed gateway port is used.

The details of flow restrictions to specific chassis are described in the **ovn−northd** documentation.

While most of the physical location dependent aspects of distributed gateway ports can be handled by restricting some flows to specific chassis, one additional mechanism is required. When a packet leaves the ingress pipeline and the logical egress port is the distributed gateway port, one of two different sets of actions is required at table 32:

- If the packet can be handled locally on the sender's hypervisor (e.g. one-to-one NAT traffic), then the packet should just be resubmitted locally to table 33, in the normal manner for distributed logical patch ports.

- However, if the packet needs to be handled on the chassis associated with the distributed gateway port (e.g. one-to-many SNAT traffic or non-NAT traffic), then table 32 must send the packet on a tunnel port to that chassis.

In order to trigger the second set of actions, the **chassisredirect** type of southbound **Port_Binding** has been added. Setting the logical egress port to the type **chassisredirect** logical port is simply a way to indicate that although the packet is destined for the distributed gateway port, it needs to be redirected to a different chassis. At table 32, packets with this logical egress port are sent to a specific chassis, in the same way that table 32 directs packets whose logical egress port is a VIF or a type **l3gateway** port to different chassis. Once the packet arrives at that chassis, table 33 resets the logical egress port to the value representing the distributed gateway port. For each distributed gateway port, there is one type **chassisredirect** port, in addition to the distributed logical patch port representing the distributed gateway port.

High Availability for Distributed Gateway Ports

OVN allows you to specify a prioritized list of chassis for a distributed gateway port. This is done by associating multiple **Gateway_Chassis** rows with a **Logical_Router_Port** in the **OVN_Northbound** database.

When multiple chassis have been specified for a gateway, all chassis that may send packets to that gateway will enable BFD on tunnels to all configured gateway chassis. The current master chassis for the gateway is the highest priority gateway chassis that is currently viewed as active based on BFD status.

For more information on L3 gateway high availability, please refer to http://docs.openvswitch.org/en/latest/topics/high-availability.

Life Cycle of a VTEP gateway

A gateway is a chassis that forwards traffic between the OVN-managed part of a logical network and a physical VLAN, extending a tunnel-based logical network into a physical network.

The steps below refer often to details of the OVN and VTEP database schemas. Please see **ovn−sb**(5), **ovn−nb**(5) and **vtep**(5), respectively, for the full story on these databases.

1. A VTEP gateway's life cycle begins with the administrator registering the VTEP gateway as a **Physical_Switch** table entry in the **VTEP** database. The **ovn−controller−vtep** connected to this VTEP database, will recognize the new VTEP gateway and create a new **Chassis** table entry for it in the **OVN_Southbound** database.

2. The administrator can then create a new **Logical_Switch** table entry, and bind a particular vlan on a VTEP gateway's port to any VTEP logical switch. Once a VTEP logical switch is bound to a VTEP gateway, the **ovn−controller−vtep** will detect it and add its name to the *vtep_logical_switches* column of the **Chassis** table in the **OVN_Southbound** database. Note, the *tunnel_key* column of VTEP logical switch is not filled at creation. The **ovn−controller−vtep** will set the column when the correponding vtep logical switch is bound to an OVN logical network.

3. Now, the administrator can use the CMS to add a VTEP logical switch to the OVN logical network. To do that, the CMS must first create a new **Logical_Switch_Port** table entry in the **OVN_Northbound** database. Then, the *type* column of this entry must be set to "vtep". Next, the *vtep-logical-switch* and *vtep-physical-switch* keys in the *options* column must also be specified, since multiple VTEP gateways can attach to the same VTEP logical switch.

4. The newly created logical port in the **OVN_Northbound** database and its configuration will be passed down to the **OVN_Southbound** database as a new **Port_Binding** table entry. The **ovn−controller−vtep** will recognize the change and bind the logical port to the corresponding VTEP gateway chassis. Configuration of binding the same VTEP logical switch to a different OVN logical networks is not allowed and a warning will be generated in the log.

5. Beside binding to the VTEP gateway chassis, the **ovn−controller−vtep** will update the *tunnel_key* column of the VTEP logical switch to the corresponding **Datapath_Binding** table entry's *tunnel_key* for the bound OVN logical network.

6. Next, the **ovn−controller−vtep** will keep reacting to the configuration change in the **Port_Binding** in the **OVN_Northbound** database, and updating the **Ucast_Macs_Remote** table in the **VTEP** database. This allows the VTEP gateway to understand where to forward the unicast traffic coming from the extended external network.

7. Eventually, the VTEP gateway's life cycle ends when the administrator unregisters the VTEP gateway from the **VTEP** database. The **ovn−controller−vtep** will recognize the event and remove all related configurations (**Chassis** table entry and port bindings) in the **OVN_Southbound** database.

8. When the **ovn−controller−vtep** is terminated, all related configurations in the **OVN_Southbound** database and the **VTEP** database will be cleaned, including **Chassis** table entries for all registered VTEP gateways and their port bindings, and all **Ucast_Macs_Remote** table entries and the **Logical_Switch** tunnel keys.

SECURITY

Role-Based Access Controls for the Soutbound DB

In order to provide additional security against the possibility of an OVN chassis becoming compromised in such a way as to allow rogue software to make arbitrary modifications to the southbound database state and

thus disrupt the OVN network, role-based access controls (see **ovsdb−server(1)** for additional details) are provided for the southbound database.

The implementation of role-based access controls (RBAC) requires the addition of two tables to an OVSDB schema: the **RBAC_Role** table, which is indexed by role name and maps the the names of the various tables that may be modifiable for a given role to individual rows in a permissions table containing detailed permission information for that role, and the permission table itself which consists of rows containing the following information:

Table Name
The name of the associated table. This column exists primarily as an aid for humans reading the contents of this table.

Auth Criteria
A set of strings containing the names of columns (or column:key pairs for columns containing string:string maps). The contents of at least one of the columns or column:key values in a row to be modified, inserted, or deleted must be equal to the ID of the client attempting to act on the row in order for the authorization check to pass. If the authorization criteria is empty, authorization checking is disabled and all clients for the role will be treated as authorized.

Insert/Delete
Row insertion/deletion permission; boolean value indicating whether insertion and deletion of rows is allowed for the associated table. If true, insertion and deletion of rows is allowed for authorized clients.

Updatable Columns
A set of strings containing the names of columns or column:key pairs that may be updated or mutated by authorized clients. Modifications to columns within a row are only permitted when the authorization check for the client passes and all columns to be modified are included in this set of modifiable columns.

RBAC configuration for the OVN southbound database is maintained by ovn-northd. With RBAC enabled, modifications are only permitted for the **Chassis**, **Encap**, **Port_Binding**, and **MAC_Binding** tables, and are resstricted as follows:

Chassis
Authorization: client ID must match the chassis name.

Insert/Delete: authorized row insertion and deletion are permitted.

Update: The columns **nb_cfg**, **external_ids**, **encaps**, and **vtep_logical_switches** may be modified when authorized.

Encap **Authorization**: disabled (all clients are considered to be authorized. Future: add a "creating chassis name" column to this table and use it for authorization checking.

Insert/Delete: row insertion and row deletion are permitted.

Update: The columns **type**, **options**, and **ip** can be modified.

Port_Binding
Authorization: disabled (all clients are considered authorized. A future enhancement may add columns (or keys to **external_ids**) in order to control which chassis are allowed to bind each port.

Insert/Delete: row insertion/deletion are not permitted (ovn-northd maintains rows in this table.

Update: Only modifications to the **chassis** column are permitted.

MAC_Binding
Authorization: disabled (all clients are considered to be authorized).

Insert/Delete: row insertion/deletion are permitted.

Update: The columns **logical_port**, **ip**, **mac**, and **datapath** may be modified by ovn-controller.

Enabling RBAC for ovn-controller connections to the southbound database requires the following steps:

1. Creating SSL certificates for each chassis with the certificate CN field set to the chassis name (e.g. for a chassis with **external−ids:system−id=chassis−1**, via the command "**ovs−pki −B 1024 −u req+sign chassis−1 switch**").
2. Configuring each ovn-controller to use SSL when connecting to the southbound database (e.g. via "**ovs−vsctl set open . external−ids:ovn−remote=ssl:x.x.x.x:6642**").
3. Configuring a southbound database SSL remote with "ovn-controller" role (e.g. via "**ovn−sbctl set−connection role=ovn−controller pssl:6642**").

DESIGN DECISIONS

Tunnel Encapsulations

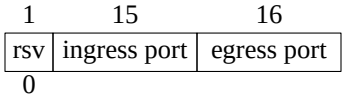
OVN annotates logical network packets that it sends from one hypervisor to another with the following three pieces of metadata, which are encoded in an encapsulation-specific fashion:

- 24-bit logical datapath identifier, from the **tunnel_key** column in the OVN Southbound **Datapath_Binding** table.
- 15-bit logical ingress port identifier. ID 0 is reserved for internal use within OVN. IDs 1 through 32767, inclusive, may be assigned to logical ports (see the **tunnel_key** column in the OVN Southbound **Port_Binding** table).
- 16-bit logical egress port identifier. IDs 0 through 32767 have the same meaning as for logical ingress ports. IDs 32768 through 65535, inclusive, may be assigned to logical multicast groups (see the **tunnel_key** column in the OVN Southbound **Multicast_Group** table).

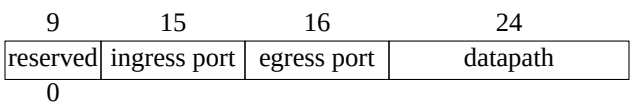
For hypervisor-to-hypervisor traffic, OVN supports only Geneve and STT encapsulations, for the following reasons:

- Only STT and Geneve support the large amounts of metadata (over 32 bits per packet) that OVN uses (as described above).
- STT and Geneve use randomized UDP or TCP source ports that allows efficient distribution among multiple paths in environments that use ECMP in their underlay.
- NICs are available to offload STT and Geneve encapsulation and decapsulation.

Due to its flexibility, the preferred encapsulation between hypervisors is Geneve. For Geneve encapsulation, OVN transmits the logical datapath identifier in the Geneve VNI. OVN transmits the logical ingress and logical egress ports in a TLV with class 0x0102, type 0x80, and a 32-bit value encoded as follows, from MSB to LSB:



Environments whose NICs lack Geneve offload may prefer STT encapsulation for performance reasons. For STT encapsulation, OVN encodes all three pieces of logical metadata in the STT 64-bit tunnel ID as follows, from MSB to LSB:



For connecting to gateways, in addition to Geneve and STT, OVN supports VXLAN, because only VXLAN support is common on top-of-rack (ToR) switches. Currently, gateways have a feature set that matches the capabilities as defined by the VTEP schema, so fewer bits of metadata are necessary. In the future, gateways that do not support encapsulations with large amounts of metadata may continue to have a reduced feature set.