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.md** 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. (Support for the latter will come later in OVN implementation.)

  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.

```
                                  CMS
                                   |
                                   |
                       +-----------|-----------+
                       |           |           |
                       |     OVN/CMS Plugin    |
                       |           |           |
                       |           |           |
                       |   OVN Northbound DB   |
                       |           |           |
                       |           |           |
                       |       ovn−northd      |
                       |           |           |
                       +-----------|-----------+
                                   |
                                   |
                         +-------------------+
                         | OVN Southbound DB |
                         +-------------------+
                                   |
                                   |
                +------------------+------------------+
                |                  |                  |
  HV 1          |                  |    HV n          |
+---------------|---------------+  .  +---------------|---------------+
|               |               |  .  |               |               |
|        ovn−controller         |  .  |        ovn−controller         |
|         |          |          |  .  |         |          |          |
|         |          |          |     |         |          |          |
|  ovs−vswitchd   ovsdb−server  |     |  ovs−vswitchd   ovsdb−server  |
|                               |     |                               |
+-------------------------------+     +-------------------------------+
```

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 create this bridge prior to starting **ovn−controller**. 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.md**) takes care of this. (This is not part of OVN or new to OVN; this is pre-existing 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. See **In−Band Control** in **DESIGN.md** 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.

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_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.md**) 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, it updates the local hypervisor's OpenFlow tables so that packets to and from the VIF are properly handled. Afterward, 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.

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_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_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_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_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_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.

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 Nicira extension register number 6. (This field is passed across tunnels as part of the tunnel key.)

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 Nicira extension register number 7. (This field is passed across tunnels as part of the tunnel key.)

conntrack zone field

A field that denotes the connection tracking zone. 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 Nicira extension register number 5.

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 16 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 16 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 15 become OpenFlow tables 16 through 31). For a given packet, the logical ingress pipeline eventually executes zero or more **output** actions:

   - If the pipeline executes no **output** actions at all, the packet is effectively dropped.

   - Most commonly, the pipeline executes one **output** action, which **ovn−controller** implements by resubmitting the packet to table 32.

   - If the pipeline can execute 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.)

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 discards packets whose logical ingress and egress port are the same.

   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 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.

Table 34 matches and drops packets for which the logical input and output ports are the same. It resubmits other packets to table 48.

4. OpenFlow tables 48 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. OpenFlow table 64 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.

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 0xffff, type 0, and a 32-bit value encoded as follows, from MSB to LSB:

1	15	16
rsv	ingress port	egress port
0		

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:

9	15	16	24
reserved	ingress port	egress port	datapath
0			

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.