OvS manipulation with Go at DigitalOcean

Matt Layher, Software Engineer
OvSCon 2017
● Software Engineer at DO for ~3.5 years
● Focused on virtual networking primitives
● Huge fan of Go programming language
● GitHub + Twitter: @mdlayher
Cloud computing designed for developers

- Cloud provider with focus on simplicity
- “Droplet” product is a virtual machine
- Compute, storage, **networking**, and monitoring primitives
  - Load Balancers as a Service
  - Floating IPs
  - Cloud Firewalls (learn more at Kei Nohguchi’s talk!)
DO is powered by Open vSwitch!

- 10,000+ of instances of OvS!
- One of the most crucial components in our entire stack.
Open vSwitch at DigitalOcean: The Past
Open vSwitch and Perl

- Events (create droplet, power on, etc.) reach a hypervisor
- Perl event handlers pick up events and performs a series of actions to prepare a droplet
- **Perl builds flow strings and calls ovs-ofctl**
my $ipv4_flow_rules = [
    # Flow priority.
    2020,
    # Hash of flow matches.
    {
        dl_src => $mac,
        in_port => $ovs_port_id,
        ip => undef,
        nw_src => "${ip}/32",
    },
    # Literal string of flow actions.
    "mod_vlan_vid:${ipv4vlan},resubmit(,1)"
],
# ... many more flows
]
# Build comma-separated matches from hash.
my $flow = _construct_flow_string($flow_hash);

# Build the flow string with usual fields.
my $cmd = "priority=${priority},idle_timeout=${timeout},${flow},actions=${actions}";

# Once a flow is added, we need a way to delete it later on!
if ($add_delete_hook && defined($delete_hook_handle)) {
    # Flows written into a libvirt hook to be deleted later.
    if (_write_flow_to_be_deleted($bridge, $delete_hook_handle, $flow) != PASS) {
        return FAIL;
    }
}

# Shell out to ovs-ofctl and do the work!
return _run_ovs_cmd("ovs-ofctl add-flow ${bridge} '${cmd}'");
Conclusions: Open vSwitch and Perl

- **Pros:**
  - Straightforward code
  - Perl is well-suited to manipulating strings

- **Cons:**
  - No sanity checking (other than OvS applying the flow)
  - Lacking test coverage
  - libvirt flow deletion hooks for individual flows proved problematic
  - Shell out once per flow; no atomicity
Open vSwitch at DigitalOcean: The Present
Open vSwitch and Go

- Events reach a hypervisor
- Perl/Go (it depends) systems perform a series of actions to prepare a droplet
- Go builds flow strings and calls ovs-ofctl
package ovs

Package ovs is a client library for Open vSwitch which enables programmatic control of the virtual switch.
package ovs

- Go package for manipulating Open vSwitch
- No DigitalOcean-specific code!
- Open source (soon)!
  - https://github.com/digitalocean/go-openvswitch
Building flows with Go

flow := &ovs.Flow{
    // Commonly used flow pieces are struct fields.
    Priority: 2000,
    Protocol: ovs.ProtocolIPv4,
    InPort:   droplet.PortID,
    // Matches and Actions are Go interfaces; functions create a
    // type that implements the interface.
    Matches: []ovs.Match{
        ovs.DataLinkSource(r.HardwareAddr.String()),
        ovs.NetworkSource(r.IP.String()),
    },
    Actions: []ovs.Action{
        ovs.Resubmit(0, tableL2Rewrite),
    },
}
Building flows with Go (cont.)

- Our example flow marshaled to textual format:

```plaintext
priority=2000,ip,in_port=1,dl_src=de:ad:be:ef:de:ad, \ nw_src=192.168.1.1,table=0,idle_timeout=0,actions=resubmit(,10)
```

- Mostly string manipulation behind the scenes; just like Perl
- Go is statically typed, reducing chance of programmer errors
- Can validate each match and action for correctness without hitting OvS
The ovs.Match Go interface

type Match interface {
    // MarshalText() (text []byte, err error)
    encoding.TextMarshaler
}

- Because of the way Go interfaces work, any type with a MarshalText method can be used as an ovs.Match
- The error return value can be used to catch any bad input
- ovs.Action’s definition is identical
The ovs.Client Go type

// Configure ovs.Client with our required OpenFlow flow format and protocol.

// $ ovs-vsctl --may-exist add-br br0
err := client.VSwitch.AddBridge("br0")

// $ ovs-ofctl add-flow --flow-format=OXM-OpenFlow14 --protocols=OpenFlow14 br0 ${flow}
err = client.OpenFlow.AddFlow("br0", exampleFlow())

● ovs.Client is a wrapper around ovs-vsctl and ovs-ofctl commands
● ovs.New constructor uses “functional options” pattern for sane defaults
● We can still only apply one flow at a time... right?
// Assume we want to apply a new flow set and remove old one.
add, remove := newFlows(), oldFlows()

// We can apply all of these flows atomically using a flow bundle!
    // $ echo -e "delete priority=10,cookie=1,actions=drop\n" >> mem
    tx.Delete(remove...)
    // $ echo -e "add priority=10,cookie=1,actions=output:1\n" >> mem
    tx.Add(add...)
    // $ cat mem | ovs-ofctl --bundle add-flow --flow-format=OXM-OpenFlow14 --protocols=OpenFlow14 br0 -
    return tx.Commit()
})

- Flow bundle stored in memory, passed directly from buffer to ovs-ofctl
- Modifications are processed by OvS in a single, atomic transaction
package hvflow

Package hvflow provides Open vSwitch flow manipulation at the hypervisor level.
package hvflow

- **DigitalOcean-specific** wrapper for package ovs
- Provides **higher-level constructs**, such as:
  - enable public IPv4 and IPv6 connectivity
  - reset and apply security policies
  - disable all connectivity
The hvflow.Client Go type

// Configure hvflow.Client to modify bridge “br0”.
client, err := hvflow.NewClient("br0", ovs.New(
))

- hvflow.Client is a **high-level wrapper** around ovs.Client
- hvflow.NewClient constructor uses “functional options” pattern for sane defaults
Network parameters - “netparams”

- Encode all necessary information about how to enable networking for a given VNIC
- Carries IPv4 and IPv6 addresses, firewall configurations, floating IPs...

- netparams used to configure OvS with hvflow.Client.Transaction method

```json
{
  "droplet_id": 1,
  "vnics": [
    {
      "mac": "de:ad:be:ef:de:ad",
      "enabled": 1,
      "name": "tapext1",
      "interface_type": "public",
      "addresses": {
        "ipv4": [
          {
            "ip_address": "10.0.0.10",
            "masklen": 20,
            "gateway": "10.0.0.1"
          }
        ]
      }
    }
  ]
}
```
hvflow.Client transactions

// Assume a netparams structure similar to the one just shown.
params, ifi := networkParams(), "public"
err := client.Transaction(ctx, func(ctx context.Context, tx *hvflow.Transaction) error {
    // Convert netparams into hvflow simplified representation.
    req, ok, err := hvflow.NewIPv4Request(params, ifi)
    if err != nil {
        return err
    }
    if ok {
        // If IPv4 configuration present, apply it!
        if err := tx.EnableIPv4(ctx, req); err != nil {
            return wrapError(err, "failed to enable IPv4 networking")
        }
    }
    return tx.Commit()
})
hvflow.Client transactions (cont.)

- Each operation accumulates additional flows to be applied within the context of the transaction.
- Flow set sizes can vary from a couple dozen to several hundred flows.

- Flows are always applied using a flow bundle; non-transactional hvflow.Client API was deleted!

```go
// IPv4 configuration.
err := tx.EnableIPv4(ctx, req4)

// IPv6 configuration.
err = tx.EnableIPv6(ctx, req6)

// Floating IPv4 configuration.
err = tx.EnableFloatingIPv4(ctx, req4F)

// Disable networking on an interface.
err = tx.Disable(ctx, 10, "public")

// Apply flow set to OvS.
err = tx.Commit()
```
The hvflow.Cookie Go interface

type Cookie interface {
    Marshal() (uint64, error)
    Unmarshal(i uint64) error
}

- Cookie structs packed and unpacked from uint64 form
- Cookies are versioned using a 4-bit identifier
- Used to store identification metadata about a flow
- Easy deletions of flows; much simpler deletion hooks with libvirt
hvflowctl and hvflowd

gRPC client and server that manipulate Open vSwitch
hvflowctl and hvflowd

- **gRPC** client and server written in **Go**
- hvflowctl passes netparams and other data to hvflowd
- hvflowd manipulates OvS flows via hvflow package
hvflowd’s gRPC interface

- gRPC uses protocol buffers ("protobuf") for RPC communication
- RPCs accept one message type and return another
- netparamspb package for encoding netparams in protobuf

```protobuf
// The set of RPCs that make up the "HVFlow" service.

service HVFlow {
  // Add flows using the parameters specified in request.
  rpc AddFlows(AddFlowsRequest) returns (AddFlowsResponse);
}

// RPC parameters encoded within a request message.

message AddFlowsRequest {
  // netparams have a protobuf representation too.
  netparamspb.NetworkParams network_params = 1;
  string interface_type = 2;
}

// No need to return any data on success.

message AddFlowsResponse {}
hvflowd AddFlows RPC

// AddFlows requests that hvflowd add flows to enable connectivity for one or more droplets.
    // Fetch netparams from gRPC request message.
    params := req.GetNetworkParams()
    // Perform the necessary transaction logic to establish connectivity.
    err := s.hvflowc.Transaction(ctx, func(ctx context.Context, tx *hvflow.Transaction) error {
        // hvflow.Client transaction logic ...
        return tx.Commit()
    })
    // Inform the caller if the request was successful.
    return &hflowpb.AddFlowsResponse{}, err
}

- RPCs enable *orchestration* among multiple hypervisors and hvflowd instances
Testing hvflowctl and hvflowd

- Unit tests verify a flow looks a certain way
  - go test ./...
- Integration tests verify the behavior of flows applied to OvS
  - mininet, OvS, hvflowctl, hvflowd
Testing with mininet

- Network topology created with multi-namespace OvS in mininet
- hvflowd spun up in each “hypervisor namespace”

```python
# Spin up hvflowd in the background and set per-hypervisor environment
# variables needed to enable multiple instances to run together.
for key in self.hvflowdEnv:
    self.cmd('export %s=%s' % (key, self.hvflowdEnv[key]))

self.cmd('%s &' % (self.hvflowdPath))

• hvflowctl issues RPCs using JSON netparams fixtures

# Issue RPCs to hvflowd using flags and netparams JSON.
self.switch.cmd("echo '%s | %s %s --rpc %s %s' % (stdinBuf, self.hvflowctlPath, command, rpcAddr, opts))
```
Testing with mininet (cont.)

- Docker image built and pulled by Concourse CI
- Virtual droplet and hypervisor topology spun up by mininet
- Tests run on **every** pull request to hvflow package

Beginning testing with /configs/production.json

===> Outside to public droplet d1 should pass
out1 (8.8.8.8) ----> d1 (192.0.2.10) *** Results: 0% dropped (1/1 received)

===> Outside to floating on d1 should pass
out1 (8.8.8.8) ----> d1 (192.0.2.100) *** Results: 0% dropped (1/1 received)

===> Outside to private d1 should fail
out1 (8.8.8.8) ----> X(d1) (10.60.5.5) *** Results: 100% dropped (0/1 received)
Conclusions: Open vSwitch and Go

- **Pros:**
  - Go is well-suited to building large, highly concurrent, network systems
  - Go compiles to a single, statically-linked binary for trivial deployment
  - Flows are easier to read for those who aren’t familiar with OvS
  - Data is statically typed and checked before hitting OvS
  - Flows can be bundled and committed atomically

- **Cons:**
  - Flows structures are verbose if you are familiar with OvS
  - We are still shelling out to OvS tools
Open vSwitch at DigitalOcean: The Future
Orchestrated Open vSwitch and Go

- RPC performed to a "regional network service" (RNS)
- RNS determines actions, sends RPCs to hvflowd
- hvflowd builds flows and speaks OpenFlow directly
Use cases for an OvS orchestration system

- High level networking actions that apply to many hypervisors and their droplets
  - Apply firewall “open TCP/22” to all droplets for customer X
  - Disable connectivity to all droplets for customer Y
Why speak OpenFlow directly?

- Difficulty parsing unstable OvS tool text output
  - `ovs-ofctl dump-ports br0 tap0`
- Tedious generation and parsing of flow strings
Why not use an OpenFlow controller?

- Industry is moving to a **distributed control plane** approach
  - Need to carefully avoid architectures where it would be difficult to scale a central controller
- We considered OVN, but were concerned about its maturity
  - The “RNS” architecture is similar to OVN
- A **distributed** OpenFlow controller is not off the table!
  - Maybe hvf1owd becomes OvS’s “controller”?
// Configure ovs.Client with our required OpenFlow flow format and protocol.
client := ovs.New(
    ovs.FlowFormat("OXM-OpenFlow14"),
    ovs.Protocols("OpenFlow14"),
    // Toggle on direct OpenFlow support.
    ovs.UseOpenFlow("localhost:6633"),
)

// Flow marshaled to binary instead of text format, and sent via OpenFlow.
err = client.OpenFlow.AddFlow("br0", exampleFlow())

- ovs.New constructor gains a new option to toggle on OpenFlow
- ovs.Client opens a socket and sends raw OpenFlow commands
**ovs.Match gains a new method**

```go
type Match interface {
    // MarshalText() (text []byte, err error)
    encoding.TextMarshaler
    // New: MarshalBinary() (bin []byte, err error)
    encoding.BinaryMarshaler
}
```

- Implement a `MarshalBinary` method for all Match types
- `ovs.Action` would be updated in the same way
Conclusions: orchestrated Open vSwitch and Go

- **Pros:**
  - Easy to orchestrate changes amongst multiple servers
  - No more parsing OvS tool string output
  - No more generating and parsing the flow text format!

- **Cons:**
  - Too early to tell!
Open vSwitch at DigitalOcean: Conclusions
DO is powered by Open vSwitch!

- We’ve deployed more than **10,000** instances of OvS and have run it in production for **three years**.
- We’ve moved from **Perl to Go**, and our OvS tooling has too.
- We’re excited for the **future** of OvS and OVN!
Thank you!

Matt Layher
mdlayher@do.co
GitHub + Twitter: @mdlayher

https://github.com/digitalocean/go-openvswitch