Intel Cloud Orchestration Networking

Spring Midterm Progress Report

Matthew Johnson, Cody Malick, and Garrett Smith

Team 51, Cloud Orchestra

Abstract

This document outlines the progress of the Cloud Orchestration Networking project for Spring 2017. It contains a short description of the project's purposes and goals, current progress, code samples, current issues, and any solutions to those issues.

CONTENTS

I. Project Goals

Our project is to first switch the Linux-created GRE tunnel implementation in Ciao to use GRE tunnels created by Open vSwitch. From that point we will switch the actual tunneling implementation from GRE to VxLAN/nvGRE based on performance measurements of each on data center networking cards. After this is completed, a stretch goal is to replace Linux bridges with Open vSwitch switch instances.

These goals changed somewhat by the middle of the Winter term. The primary goal now is to replace the Linux bridges with Open vSwitch switch instances because of an assumption that was found to be incorrect. It was assumed that we could create tunnel endpoints with Open vSwitch without using Open vSwitch bridges but Open vSwitch could not create tunnel endpoints with the Linux bridges Ciao uses. A full integration of Open vSwitch was required to use Open vSwitch created tunnels. Initially, we had planned on using a third party API, `libovsdb` to interface with the Open vSwitch management database [1]. While providing the necessary functionality, it added undocumented overhead. Specifically, all bridges and tunnels generated by Ciao had to be known about in the calling library. After extensive research and discussion with our client, we aimed to fully implement Open vSwitch into Ciao, rather than use it to exclusively create tunnels.

This scope change pushed the goal to switch the tunneling implementation to VxLAN/nvGRE based on performance measurements to stretch goal status. All scope change details were approved by our client.

II. Purpose

The current implementation of Ciao tightly integrates software defined networking principles to leverage a limited local awareness of just enough of the global cloud's state. Tenant overlay networks are used to overcome traditional hardware networking challenges by using a distributed, stateless, self-configuring network topology running over dedicated network software appliances. This design is achieved using Linux-native Global Routing Encapsulation (GRE) tunnels and Linux bridges, and scales well in an environment of a few hundred nodes.

While this initial network implementation in Ciao satisfies current simple networking needs, all innovation around software defined networks has shifted to the Open vSwitch (OVS) framework. Moving Ciao to OVS will allow leverage of packet acceleration frameworks like the Data Plane Development Kit (DPDK) as well as provide support for multiple tunneling protocols such as VxLAN and nvGRE. VxLAN and nvGRE are equal cost multipath routing (ECMP) friendly, which could increase network performance overall.

III. Spring Progress

Over spring break, and through the first few weeks of the term, we were in active development on the project. Working towards a working prototype, we were aiming to fully flesh out the Open vSwitch interfaces we had created. This involved altering the `bridge` object that Ciao uses to track network connections. Specifically, updating the bridge to support various network modes, and adding the appropriate logic to utilize the new mode.

A very important piece of code is the `bridge` object. The `bridge` object represents how Ciao tracks network bridges internally. This being a key aspect of our project, we altered it in a very simply way. We added a `mode` field to the `attrs` (attributes) struct. The mode field simply allows the code differentiate between the default Linux bridge network mode, and our newly implemented OVS mode.

Listing 1. The Ciao `bridge` object, and the newly added `mode` field in the `attrs` struct

```
// Bridge represents a ciao Bridge
type Bridge struct {
   Attrs
   Link *netlink.Bridge
}

// Attrs contains fields common to all device types
type Attrs struct {
   LinkName string // Locally unique device name
   TenantID string // UUID of the tenant the device belongs to
   // Auto generated. Combination of UUIDs and other params.
   // Typically assigned to the alias
   // It is both locally and globally unique
   // Fully qualifies the device and its role
   GlobalID string
   MACAddr *net.HardwareAddr
   Mode NetworkMode
}
```

Other core code snippets are in the `ovs_bridge.go` and `ovs_gre.go` files. These files contain the primary interfaces we designed and implemented to allow Ciao to call the `ovs-vsctl` command line utility. As per our scope change, we had to drop the use of the `libovsdb`, the third party library we were using. Having to use OVS in the full networking stack, both bridge and tunnel rather than exclusively tunnel, means that Ciao has to have a way of calling the OVS management database. For this task we opted to use `ovs-vsctl`.

In order to call `ovs-vsctl`, we have to build a command in Ciao, and then use a built in Golang function called `exec()`. The `exec()` function simply allows an array of strings to be translated and executed in a shell. This made our lives very simple as we could simply build a generic function to handle dishing a command to shell, and then specific functions for all the functionality we needed. Our generic function, named `vsctlCmd()`, takes in a string and calls `exec()`.

Listing 2. The `vsctlCmd` function takes a string array, and calls the `exec` function on the string array

```
func vsctlCmd(args []string) error {
   _, err := exec.Command("ovs-vsctl", args...).Output()
```

```
    if err != nil {
        glog.Error("vsctlCmd failed: " + err.Error())
        return err
    }

    return nil
}
```

For specific functionality, we defined the commands needed from `ovs-vsctl` in individual functions. An example of this is the `createOvsBridge()` function. The `createOvsBridge()` function takes in a `bridgeId` string, and builds a command into a string array. The base of the command is the `ovs-vsctl` parameter `add-br`. After the base command, it simply adds the bridge ID to the array, and adds the remaining parameters. These types of functions make up the interfaces to create, destroy, and link our bridges and tunnels in Ciao.

Listing 3. The `createOvsBridge` function takes a string `bridgeId`, builds a command, and calls the `vsctlCmd() function`

```
func createOvsBridge(bridgeId string) error {
    // Example: ovs-vsctl add-br ovs-br1
    args := []string{"add-br", bridgeId, "--", "set", "bridge", bridgeId,
    "datapath_type=netdev"}

    // Execute command
    if err := vsctlCmd(args); err != nil {
        return err
    }

    return nil
}
```

After fully implementing the mode logic, we had a major breakthrough week two of the term. We were able to create a working VM instance in Ciao-Down, our single machine development environment. Though a major milestone, we found after some testing that we could not ssh into the machine.

Fig. 1. Instance is can be pinged but ssh is refused.

```
root@singlevm:/home/garrett/go/src/github.com/01org/ciao/testutil/singlevm# ciao-cli instance
 list
# UUID                                 Status Private IP SSH IP         SSH PORT
1 f42d8895-af47-471a-b9e9-395f5c619bb9 active 172.16.0.2 198.51.100.100 33002
root@singlevm:/home/garrett/go/src/github.com/01org/ciao/testutil/singlevm# ping -c 4 198.51.
100.100
PING 198.51.100.100 (198.51.100.100) 56(84) bytes of data.
64 bytes from 198.51.100.100: icmp_seq=1 ttl=64 time=0.378 ms
64 bytes from 198.51.100.100: icmp_seq=2 ttl=64 time=1.24 ms
64 bytes from 198.51.100.100: icmp_seq=3 ttl=64 time=1.23 ms
64 bytes from 198.51.100.100: icmp_seq=4 ttl=64 time=1.39 ms

--- 198.51.100.100 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3002ms
rtt min/avg/max/mdev = 0.378/1.062/1.391/0.402 ms
root@singlevm:/home/garrett/go/src/github.com/01org/ciao/testutil/singlevm# ssh -i ~/local/te
stkey -p 33002 demouser@198.51.100.100
ssh: connect to host 198.51.100.100 port 33002: Connection refused
```

We continued to work on this problem until recently by trying to manipulate how the controller nodes manage their virtual network interface cards (VNICs). Our idea was that the VNICs could be made to communicate with the Open vSwitch network objects if tracked correctly. This approach only broke our implementation even more. We ended up rolling back our codebase to the point where we could ping the instances but the ssh connection was refused.

IV. Issue

After consulting with Manohar Castelino, a Principle Engineer at Intel and original author of Ciao networking, he suspected that our project was very close to working. He suggested that we take a look at the maximum transmission unit, or MTU, that Ciao was configured to use. MTU is the defined size of a packet or IP frame that can be sent over the network [2]. This is configured on a per-device basis. After standardizing MTU size within the ciao-down environment, the results did not change. At present, our code is being reviewed by Manohar, and we are working with him to figure out the problem.

V. Remaining Steps

Our next steps are to work with Manohar to fix any further feedback he has for us. As mentioned above, we are able to ping created instances, but not able to ssh into them. As far as implementation goes, our sole remaining step is to fix the ssh issue. The stretch goal of doing measurements on nvGRE and VxLAN are untenable at this point in the term.

Beyond code, our next step is to present the project at the Engineering Expo on May 19th.

References

[1] Socketplane. (2016, dec) libovsdb. [Online]. Available: https://github.com/socketplane/libovsdb/blob/master/README.md

[2] M. Rouse. (2006, September) Maxium transmission unit. [Online]. Available: http://searchnetworking.techtarget.com/definition/maximum-transmission-unit