An Elastic Distributed SDN Controller

[222 Course Project Proposal]

Guo Li
A53071472
gul027@eng.ucsd.edu

Xinyu Zhang
A53095838
xiz368@eng.ucsd.edu

Liqiong Yang
A53076313
liy007@eng.ucsd.edu

1. INTRODUCTION

Software Defined Networking (SDN) provides a centralized control panel, which allows the network to be programmed by the application and controlled from one central entity, also brings the issue of reliability and scalability. Recent researchers have explored architectures for building distributed SDN controller, but all have an implication that the mapping between a switch and a controller is statically configured. With this design, it is difficult for the control plane to adapt to traffic load variation. There could be the case where a controller may become overloaded, if the switches mapped to this controller suddenly received a large number of traffic, while other controllers remain idle.

There should be a migration logic that, when load imbalance occurred, it can migrate a switch from a heavily-loaded controller to a lightly-loaded one. There also should be a monitoring logic that can decide when to trigger this migration. OpenFlow currently doesn't support such a migrate operation and this is what we want to implement in our project.

2. APPROACH

The idea came from *2013 SIGCOMM* workshop paper, 'Towards an Elastic Distributed SDN Controller', where the author built a prototype system, **ElastiCon**.

ElastiCon implemented a distributed controller schema, where one switch connects to multiple controller nodes, one of which acts as the master and the rest as slaves. Different controller nodes coordinates with each other to elect as a master when migration happened. ElastiCon used distributed data store, Hazelcast, to glue the cluster of controllers to provide a logically centralized controller. It store all switch information that is shared among the controllers.

The system designed a switch migration protocol, which can guarantee the **liveness**, at least one controller is active for a switch at all times, and **safety**, exactly one controller processes every asynchronous message from the switch, during a migration. The system also provided load adaptation logic check the load and trigger the migration event.

We plan to to use Mininet and Python RYU, first implement our own version of ElastiCon, and then validate the result of this research work.

For the evaluation, we will focus on the control plane traffic load instead of the data plane, also try to simulate larger network using multiple VMs. It should be not hard to simulate the packet exchange. But since the overhead of emulating data flows, according to the paper, we may also need to modify Open vSwitch to inject Packet-In messages to the controller without actually transmitting packets on the data plane. We also should log and drop Flow-Mod messages to avoid the additional overhead of inserting them in the flow table. We all find this part of paper is not very easy to understand, so we will keep digesting and try to give detailed explanation based on the results we will get throughout the project.

3. RELATED WORK

The SDN controller architecture has evolved from the original single-threaded design to the more advanced multi-threaded design in recent years. Although the single-controller system become more and more efficient, it still has limits on scalability and vulnerability. Recent research has explored the design of distributed controller system. Onix uses a transactional database for persistent but less dynamic data, and memory-only DHT for data that changes quickly but does not require consistency. Hyperflow replicates the event at all distributed nodes, so each node can process such events and update their local state. D. Levin, etc, has further elaborated the state distributed trade-offs in SDN.

All existing distributed controller designs implicitly assume static mapping between switches and controllers, and hence lack the capability of dynamic load adaptation and elasticity.

4. PLAN OF ACTION

1. Learning
2. Setup Mininet Environment in VMs, and install Python RYU, try to work with Mininet through RYU OpenFlow Libary
3. Implement the Naive Version of the Protocol
4. Implement the Protocol Proposed in the Paper

5. Evaluate and Collection Results, and Analyze, and compare the Pros and Cons between Naive version and Proposed Version

6. Try to optimize the system so the evaluation will have more realism. New ideas to change the protocol are welcomed to test

7. Final Analysis and Result Formalizing

5. SCHEDULE

Our primary mission now is to get familiar with OpenFlow Spec and some tools and libraries, start to setup development and runtime environment in the VMs and probably our machines, too. In the meantime, reach deep understanding for this paper.

By the first checkpoint 11/3, which is ten days from now. We plan to finish all the setup, get enough knowledge and simulate a basic SDN with tools in hand. Moreover, we need to provide the detailed technical plan about what should we modify the basic codebase to reach our goal.

By the second checkpoint 11/24, we have to implement the model completely and already have a lot of evaluation results in hand. It should support to simulate large networks. In the same time, we will be trying different evaluation methods, and optimize the implementation to get more accurate results.

By the end, we should wrap up all the work and get the poster and paper done.