Fault Tolerance in Software Defined Networking

Suresh Gururajan
MS in Computer Engineering
Arizona State University
Tempe, AZ, USA
Suresh.gururajan@asu.edu

Ponneeswaran Natarajan
MS in Computer Science
Arizona State University
Tempe, AZ, USA
Pnatara8@asu.edu

***Abstract*— An important property of Software Defined Networking (SDN), often overlooked or not considered, is fault tolerance which defines reliability and availability of the network. We explored the concepts of fault tolerance and scenarios where a network failure can occur. We identified a set of common metrics, such as flow switchover time, number of packets dropped and controller switchover time, to evaluate the fault tolerant system. The goals of this project are to simulate a simple fault tolerant system, that rapidly recovers from a network failure and evaluate the metrics we identified.**

Keywords—fault tolerance, SDN, reliability, availability, switchover

I. INTRODUCTION

Software Defined Networking introduces the separation of control plane from the data plane. SDN enables management of network services by providing abstraction of lower-level functionality. In the control plane, a controller (logically centralized) is responsible for making traffic forwarding decisions, while in the data plane, a switch is responsible for packet forwarding. So SDN simplifies development of network applications.

However, an important property of SDN, fault tolerance is often overlooked. According to Hyojoon et. al [1], "…there has been little discussion on how to deal with an age-old yet common problem in computer networks: network failures" [p. 1]. For example, link disconnections can occur at any point of an instance and disrupt normal traffic forwarding between two nodes, two switches, or between a switch and a controller. This link downtime will result in data packet loss, which is not desirable.

A. Motivation

The motivation behind exploring fault tolerance is the rising need for network state consistency. For example, controller applications (such as load balancer) assume that the controller is aware of the network-wide state and that this view of the controller is consistent. This means that an inconsistency in this network-wide state can lead to issues in controller applications such as lower performance (or incorrect behavior like improper load balancing).

B. Scenarios

[1] classifies three areas where faults can arise. Firstly, the Ethernet links between a node and a switch can go down – this is basically a failure in the data plane. Second, the secure channel between switch and controller can go down, which is a control plane failure. Thirdly, the machine where the controller is running can fail. In this project, we try to simulate these data plane failures and controller machine failures, by manually bringing down the Ethernet links down or by stopping the controller machine. By simulating these failures we try to evaluate the metrics.

II. RELATED WORK

Desai et. al. [2] introduce the importance of having fault tolerance in control planes. They stress the importance of switches proactively taking up the role of identifying link failures rather than having the controller taking up this responsibility. Kim et. al [1] present CORONET, a controller that is tolerant to the three types of faults that can occur in a network which we described earlier in the scenarios. Reitblatt et. al. [3] take a different approach and design FatTire, a programming language, based on regular expressions for defining rules. The advantage of using FatTire is that it is nondeterministic and adapts well to changes (such as link failures) dynamically. Kempf et. al.[4] propose a monitoring function implemented on OpenFlow Switches where the switches emit link monitoring messages which can detect whether a link is down. In [5], Botelho et. al, present a distributed and consistent data store for implementing a fault tolerant control framework.

III. EXPERIMENTAL SETUP

A. Architecture

We plan to use the architecture given in Figure 1 because we feel that it accurately describes the scenarios we try to simulate in our project and helps us understand the challenges in fault tolerance better.

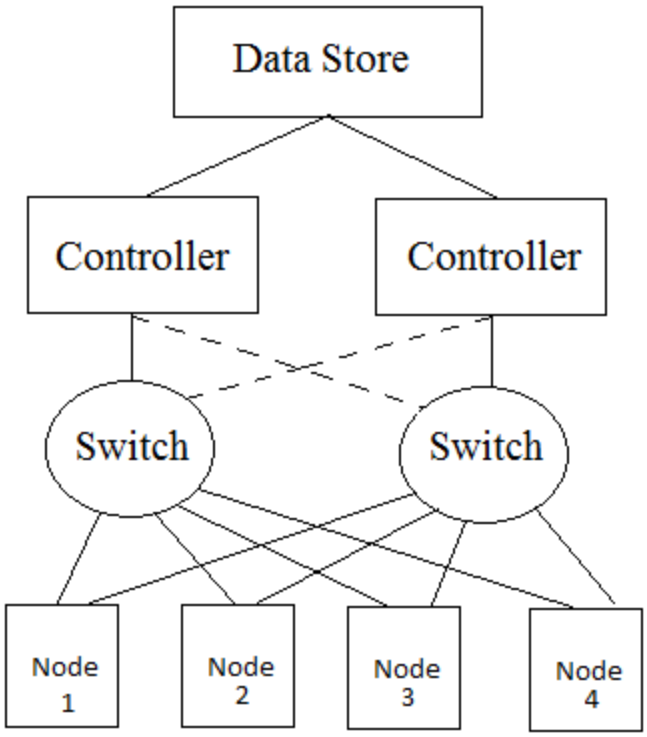

Figure 1: Architecture. The distributed controllers are connected to a central data store. The switches are connected to both controllers in master-slave configuration.

This architecture is similar to the architecture defined by Botelho et. al [5]. In this architecture, a data store is a layer of persistency which contains 1) decisions made by control plane applications based on events triggered by the OpenFlow switches and 2) consistent network-wide state (such as topology information). The controllers share the data store thereby ensuring data consistency between the two controllers. Switches are connected to controllers in master-slave configuration – by this, we mean that initially, a "master" controller handles flow decisions asked by the switch while the slave controller is in an idle state. In case of (master) controller failure, the slave controller takes over the role of master controller and handles flow decision making for the switch.

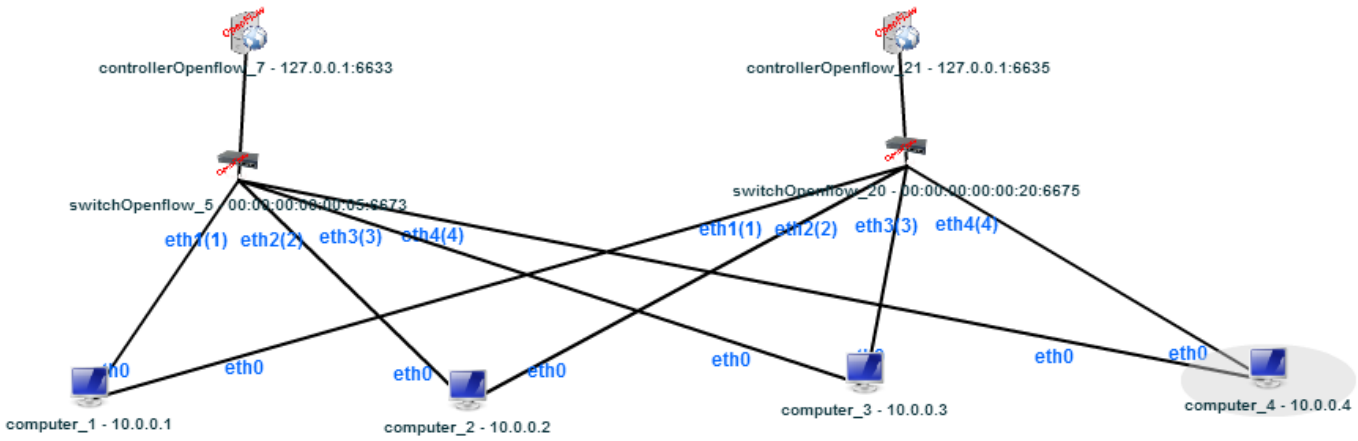

Figure 2: Topology Setup as viewed in VnD GUI.

B. *Implementation*

i. *Topology - Mininet*:

We used VnD (Virtual Network Description, SDN version by Ramon Fontes) topology generator which enables the programmer to graphically visualize the network structure and configure the components. We exported the topology setup to python script. This python script is then used to generate the topology in Mininet virtual network on a single machine. Figure 2 represents the topology setup using the VnD topology generator.

Mininet is an Open Source Licensed API for simulating a network virtually enabling developing and research. Mininet uses lightweight virtualization to make a single system look like a complete network. The topology consists of two Open Flow switches and four Linux systems. Every node is connected to both switches using Ethernet connections. In order to simplify the connections and resolve the ip address conflicts, the two Ethernet links connected in every node is assigned 0.0.0.0 ip address and a bridge, between the two Ethernet connections, is created with appropriate ip address. This ensures that the traffic packets reaches the target node regardless of the routing path. Two FloodLight controllers are run to handle the two OpenFlow switches.

ii. *FloodLight controller:*

Two java based Open SDN controllers are run at parallel and the two available OpenFlow switches are linked to them using the ovs-vsctl set-controller command. The main obstacle faced here is running two similar controllers in the same VM. In order to resolve the conflict we had to run the controllers from two different locations with modified controller resource files and Java files. The properties files are modified such that the controllers now run on different port numbers. The modified files includes property files like Neutron. properties, FloodLightDefault. properties, LearningSwitch. properties, and Java files like Controller. java, FallbackCCProvider. java, RestAPIServer .java, and server files like DebugServer.py. The controller applications are built as a Java modules, compiled with floodlight and the applications built over floodlight rest service API.

iii. *Java WatchDog Service:*

A WatchDog is defined as a Java technology-based service which launches and monitors the system manager and connectors. The WatchDog service we have written is run separately for the controller and the switches.

a. Rest Service:

The controller WatchDog service monitors for any controller failure. It checks every second for any failure. In case of any failure, the WatchDog Rest Service then writes the new updated topology to a flat file.

b. Switch WatchDog Service:

The WatchDog service, running on the switches, monitor for any controller failure or control plane failure. In case of any failure this service then changes the controller assigned to the switch to the other available controller.

iv. *Network Traffic Generation:*

We use standard ping command to reach a target node from any starting node. Since we evaluate the state of the network and its components, we find ping command sufficient.

IV. EVALUATION & RESULTS

A. *Metrics*

i. *Flow switchover time (FST)*

We define this metric as the time taken by the controller for setting a new (possible) route for a flow after the switch reports that a link is down for the flow. To explain further, the switch initially reports to the controller that a link is down. The controller evaluates the topology and calculates a new route. The controller then updates this route in the data store and sends back the flow decision to the switch. This time taken by the controller is what we define as the flow switchover time.

ii. *Number of packets dropped*

This metric denotes the number of packets dropped as a result of link failure. This is calculated from the time when the link went till a new flow decision is given by the controller. This metric is applicable for control plane failure as well. Here, we measure the number of packets dropped from the time a controller goes down till another controller takes over. We look at ping results and note down how many packets have been dropped.

iii. *Controller switchover time (CST)*

When there is a failure in the controller machine, we calculate controller switchover time. This can be evaluated by calculating the time taken for the change of a switch's controller from "master" to "slave" (the time taken for slave to become master controller). We define CST as the time taken for the duration of the following process: 1) controller machine goes down when there are flows currently going on and the switch is in "secure" mode 2) another controller ("slave" in our scenario) in our topology which periodically interacted with the current "failed" controller finds this out and sends a message to the switch (which was connected to the failed controller) asking the switch to attach itself to this "slave" controller 3) the switch uses ovs-vscctl command [refer] to attach itself to this controller. 4) The flow decision for the ongoing flow is made by the new controller. The time taken for this entire process is what we define as CST.

B. Results

We were able to handle the failure scenarios described earlier. For data plane failure, we had initially implemented a module DataPlaneWatchdog which implements the method isReachable of Java's built-in InetAddress class which in turn issues ICMP requests to the target host. When know the state of a link by periodically pinging the target host. In case of switch-host link failure, we used Floodlight's REST API to insert a static flow (after calculating an alternate route, assumed known) from the controller. We learnt that this method is basically a wrapper function of ping command which the Floodlight controller was using internally. We found out that Floodlight contains similar functionality because the flows were getting redirected before we could our static flow insertion command. Hence, we decided against integrating this module with the controller because it adds an overhead of additional calculations for our topology as the controller was intelligent enough to handle data plane failures for our topology pretty well.

In case of control plane link failures, we found that the controller switchover happens very quickly with minimal to zero packet loss, which is desirable. This happened because of two reasons: 1) we use Floodlight's lightweight REST API which fetches us the state of a controller very quickly and 2) the switch always listens to a port which tells the switch the controller it needs to attach itself to. In our project, the controller switchover process actually requires only the exchange of a port number between the controller and switch, which is lightweight.

We found varying results for the ping response time though. We assumed that this would also be uniform like packet loss, but we found that the response time varies by a larger margin. The results obtained by running our modules in case of 10 iterations is given in the graph. We issued ping commands and obtained the results of number of packets skipped and response time.

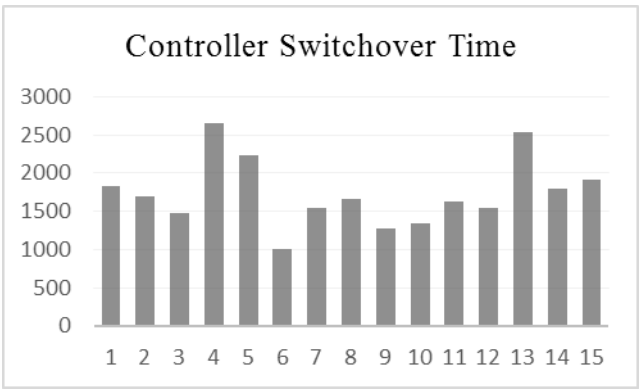

Figure 3: Control Switchover Time. X-axis: Iteration number. Y-axis: Time in Milliseconds.

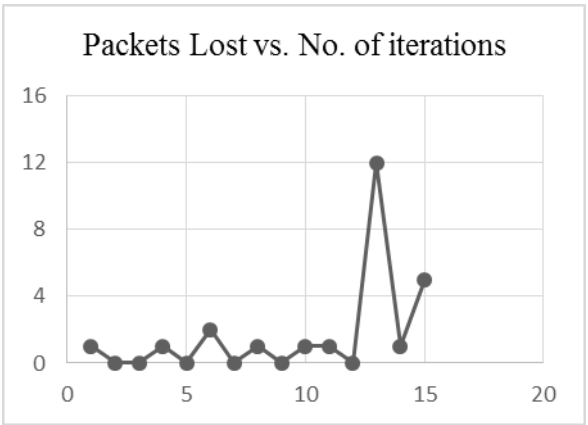

Figure 4: Number of Packet Lost. X-axis: Iteration number. Y-axis: Number of packets lost.

In our opinion, this occurs because of the timeout periods of the flow. One flow may timeout sooner than another before the controller goes down, as a result of which the latter might experience higher response time because the installed flow needs to expire from the switch. The spike shown in the graph is because we started the slave controller at almost the same time as the master controller went down. This implies that the controller startup time also has an impact on the number of packets lost. If we exclude this, the number of dropped packets falls mostly within 0-4 range. This shows that the number of packets lost is acceptable. One of our future goals is to further minimize this number.

V. FUTURE WORK

From our architecture, it is evident that the data store is a single point of failure. Since we focused on data failure and control failure, we aim to take forward the concept of data store failure too. We are working on implementing a simple fail-safe architecture with Apache Zookeeper which provides distributed systems services such as master election, synchronization, server state (up or down), etc. In this project, our data store was a file system on a single server. We aim to make this redundant (across multiple servers) and use Zookeeper to manage tasks related to distributed systems.

In addition, we aim to implement a lightweight perceptron based learning algorithm which predicts when a node (controller or host) might go down and changes the controller of a switch accordingly. Our goal is that a switch is attached to a controller which is most likely alive at a point in time. We already proved that the number of dropped packets is within the acceptable range from the graphs so it makes sense to switchover the controller, though hopefully not often.

Thirdly, we aim to take into consideration, various topologies from the topology zoo and extend our architecture to be more generic, and see if there are other metrics that need to be considered when designing a fault-tolerant SDN architecture.

REFERENCES

[1] Hyojoon Kim, Jose Renato Santos, Yoshio Turner, Mike Schlansker, Jean Tourrilhes, Nick Feamster, CORONET: Fault Tolerance for Software Defined Networks, 2012 IEEE

[2] Maulik Desai, Thyagarajan Nandagopal, Coping with Link Failures in Centralized Control Plane Architectures, 2010 IEEE

[3] Mark Reitblatt, Marco Canini, Arjun Guha, Nate Foster, FatTire: Declarative Fault Tolerance for Software-Defined Networks, HotSDN'13, August 16, 2013, Hong Kong, China

[4] James Kempf, Elisa Bellagamba, András Kern, Dávid Jocha, Attila Takacs, Pontus Sköldström, Scalable Fault Management for OpenFlow

[5] Fábio Botelho, Fernando M. V. Ramos, Diego Kreutz, Alysson Bessani, On the feasibility of a consistent and fault-tolerant data store for SDNs.