Dynamic Controller Deployment in SDN
Marc Huang, Sherrill Lin, Dominic Yan
Department of Computer Science, University of Toronto

Table of Contents
Introduction ................................................................. 1
Background and Motivation ............................................. 1
Problem Statement ......................................................... 2
Algorithm ....................................................................... 2
System Description ......................................................... 4
Evaluation and Experiment ................................................ 4
Discussion ..................................................................... 7
Summary and Future Works ............................................... 7
Task Division ................................................................. 8
References ..................................................................... 8

Introduction
In Software Defined Network (SDN), control plane is decoupled from forwarding plane, hoping to centralize network control and facilitate dynamic configuration and policy deployment among the whole network. In the traditional SDN architecture, a single controller that contains the global view of the network topology is responsible to run control plane. However, in large-scale network like data center, this implementation leads to limitations in terms of controller performance and network scalability. Firstly, a controller can hardly deal with huge traffic flows initiated within the network due to its restricted processing capacity. When the CPU usage reaches the maximum value, some flows could be discarded without processing. Secondly, the bandwidth of the controller-and-switch links can result in traffic congestion if large volume of traffic generates among the switches. Both of these factors reduce the scalability of the network so that multiple-controller architecture is introduced.

Although deploying more controllers can overcome the aforementioned limitations, new problems are raised, such as how many controllers are required and how to decide the controller placement in the network. Moreover, the controller deployment should be adjusted dynamically according to the instantaneous traffic loading of each switch. To address these issues, we propose a dynamical controller deploying framework which can modify the number of controllers and controllers placement scheme based on the topology size (in terms of number of switches) and the flow loading of switches. We hope to balance between the controller performances (in terms of flow setup time) and the controller resource utilization.

When designing the deployment framework, we mostly consider the scalability, feasibility and performance of our scheme. To simplify the solution scope and facilitate testing setup, we make the following assumptions: 1) all the switches in the system are OpenFlow enabled switches; 2) in our solution, there is only one kind of generic controllers which only possess local views of the network topology within their reign; 3) the only function of the controllers is to help switches handle layer 2 learning and forwarding; 4) the up-and-down events in the switch links are neglected so that the network topology of the nodes is static; 5) the switches function properly only when they are assigned to a live controller. The controllers can exchange switch information with each other to obtain the global view of the network topology, but the protocol of inter-controller communication is out-of-scope for this project.

Background and Motivation
The early age Software Defined Network (SDN) implementation relied on a centralized controller that possesses a global view of the network. However, this centralized approach has several limitations related to performance and scalability, especially in dealing with large amount of flows. To ensure a user-acceptable flow set-up latency and system scalability in highly demanding networks, there have been a number of techniques proposed and
experimented. Generally, these techniques can be classified into two broad categories: 1) pushing intelligence into the switch to offload the controller, and 2) distributing the control plane across multiple ‘local’ controllers, like Kandoo [4]. However, in the second category, the number of ‘local’ controllers and the placement of these controllers rely on network operators’ manual configuration, i.e. these assignments are static and requires human intervention if changes are required. Being highly motivated by this shortage, we want to propose a solution to automatically adjust the number of ‘local’ controllers and controllers-to-switches assignment according to the dynamic behavior of the network. A few studies have been carried out regarding how many controllers to place and where to place them in the network. For instance, Bari et al. [5] analyzed the impact of placing multiple controllers according to different heuristics either in a static setting or with some periodic metrics feedbacks. Based on previous studies, we put forward our solution in this project. The focus of this project is to make decision on controller-switch association based on the nearly real-time metrics feedbacks. In addition, to make the solution more accessible to users, the proposed solution grants flexibility to users in terms of customizing the coefficients of each metric in the controller performance evaluation function.

**Problem Statement**

This project mainly focuses on how to evaluate whether the current controller deployment is suitable according to the dynamic traffic statistics, and how to optimize the resource utilization by arranging the controller-to-switch association. We design a linear function to evaluate the controller performance based on the real-time collected data from the current controller deployment. Such evaluation process considers a set of metrics constraints. Also, we propose reassignment algorithm to speed up load balancing among the controllers. Furthermore, a failover mechanism is designed to address controller failure issue.

We adopted an undirected network topology $G (S, E)$, where $S$ denotes the set of switches and $E$ is the set of edges in between. Let $L$ represents the set of active controllers and $N$ is the number of controllers, which is 1 by default. $X = \{x_{ik}\}_{i \in S \times N}$ denotes the assignment relations between switches and controllers, in which every entry $x_{ik} = 1$ if switch $i$ connects to controller $k$ and $x_{ik} = 0$ otherwise. For the collected statistics, $t_{kn}$ represents the processing time for event $n$ handled by controller $k$ and $\lambda$ is the average number of flows requiring for setup at switch $i$ in current time.

**Algorithm**

**Constraints** Our objective is to minimize the number of controllers $N$ considering a series of constraints, including:

$\forall \ k \in L, C_k < C_{max}$ \hspace{1cm} (1)

$\forall \ k \in L, M_k < M_{max}$ \hspace{1cm} (2)

$\forall \ k \in L, E_k < E_{max}$ \hspace{1cm} (3)

$\forall \ k \in L, D_k < D_{max}$ \hspace{1cm} (4)

$\forall \ k \in L, \frac{\sum t_{kn}}{E_k} < t_{max}$ \hspace{1cm} (5)

$\forall \ i \in S, \sum_k x_{ik} = 1$ \hspace{1cm} (6)

For each controller $k$, equations (1) to (5) show the upper bounds for CPU usage, memory usage, average number of handled events per second and average number of dropped packets and average processing time for handled packets. More specifically, the missing packets include both packets dropped by the controller and by the associated switches. Packet dropped by controller is convincing cue of controller overloading. Under our assumption that no ups-and-downs of switches are considered, the packet dropped by the associated switches also indicates that the controller fails to handle the flow setup events. Equation (6) verifies whether every switch is assigned to one and only one controller.

**Evaluation function** In order to evaluate the utility of a controller, an evaluation function is defined as a function of the 5 metrics for each controller:

$U_k = f (E_k, C_k, M_k, D_k, t_{kn})$ \hspace{1cm} (7)

Each of the 5 metrics reflects the controller performance. In our project, we design the evaluation function as follows though there is more than one possible expression.

$U_k = \alpha E_k + \beta C_k + \gamma M_k + \mu D_k + \omega \frac{\sum t_{kn}}{E_k}$ \hspace{1cm} (8)

In this formula, $\alpha, \beta, \gamma, \mu$, and $\omega$ are coefficients that can be customized to modify the relative significance of the 5 metrics. $U_k$ is normalized to $[0, 1]$ aiming to simplify the calculation. The normal range of $U_k$ is denoted as
\[ U_{lower}, U_{upper} \] , where \( U_{lower} > 0 \) and \( U_{upper} < 1 \). When \( U_k > U_{upper} \), it means controller \( k \) is overloaded and subordinate switches should be reassigned to other controllers. If no active controllers have enough capacity (in terms of packets processing quantity), a new controller will be activated to take over the unassigned switches. When \( U_k < U_{lower} \), it means the associated switches of controller \( k \) can be merged to those of other controller to cut down resources.

Considering the constraints, once the value of a metric goes beyond the normal region, the value of the evaluation function should notice abnormal automatically. Thus, the evaluation function can be modified as follows:

\[
U_k = U_k^* \mid (E_k > E_{max}) \mid (C_k > C_{max}) \mid (M_k > M_{max}) \mid (D_k > D_{max}) \mid (\sum t_{kn}/E_k > t_{max}) \tag{9}
\]

where \( U_k^* = \alpha E_k + \beta C_k + \gamma M_k + \mu D_k + \omega \sum t_{kn}/E_k \) and \( \mid \) is the logical operator OR. If any metrics violate the constraints, \( U_k \) will become 1 immediately and trigger the centralized scheduler program.

**Reassignment** In the centralized scheduler program, for each controller, the associated switches are sorted according to their traffic loading. Meanwhile, controllers are sorted in terms of their available capacity. The largest loading switches of the overloaded controller will be first reassigned to other active controller which contains the largest available capacity (i.e. packet processing potentiality), aiming to balance the loading among controllers in a shorter time.

**Algorithm 1** Reassignment algorithm

**Input:** Network topology \( G \), Switch set \( S \), Active controller set \( L \), Traffic set \( A = [\lambda_s]_{|S|} \), Previous assignment \( X \)

**Output:** New assignment \( X^* \)

1. \( L_v \leftarrow \) List of the outstanding controllers for \( X \)
2. **for each** \( l \) from \( L_v \) **do**
3. \( H_c \leftarrow \) Sorted list (descending capacity) of controller excluding \( l \)
4. \( H_s \leftarrow \) Sorted list (descending loading) of the switches for controller \( l \)
5. \( s \leftarrow \) First switch in \( H_s \)
6. **while** statistics of \( l \) violate the constraints (1) to (6) **do**
7. \( k \leftarrow \) First controller in \( H_c \)
8. **if** \( \lambda_s < E_{max} - E_k \) **then**
9. update \( X \leftarrow \) reassign switch \( i \) to controller \( k \)
10. recalculate \( H_c, H_s \)
11. **else if** \( s \) is the last item in \( H_s \) **then**
12. update \( X \leftarrow \) add a new controller
13. recalculate \( H_c \)
14. reset \( s \leftarrow \) first switch in \( H_s \)
15. **else if** \( H_s = \emptyset \) **then**
16. update \( L_v, L_v \leftarrow \) remove \( l \)
17. break
18. **else**
19. \( s \leftarrow \) Next switch in \( H_s \)
20. **end if**
21. **end while**
22. **end for**
23. \( X^* \leftarrow X \)

In the current stage, \( C_{max} \) is decided to be 90% and \( M_{max} \) is 5MB. Other constraint values will be determined according to different scenarios.

**Failover** In the failover mechanism, statistics collection program periodically checks the heartbeat of each controller as well as the association status of each switch in every time unit. Once it detects inactivate controllers or unassigned switches, it will triggered centralized scheduler program to handle the problem. The problem is tackled in two steps: 1) scan for any available controllers among the actives and associate the unassigned switches to them; 2) if no available controllers found, boot up a new controller to take over the switches. The association process is similar to the reassignment process, i.e. the switches with heaviest loading and controllers with largest capacity have the highest priority.

In our scenario, all the controllers possess merely local views of the network topology. Since we assume the controllers in our scenario only need to handle layer 2 learning and forwarding issues, they can learn MAC address automatically and provide data-link layer forwarding. Therefore, after the reassignment process, controllers can be aware of the topology change and adapt accordingly. For layer 3 forwarding, controller should take steps to learn about the new topology whenever reassigning a switch, which will be considered in the next stage.
System Description

In our framework, there are four main components: OpenFlow topologies on Mininet [1], POX controllers [3], Statistics collection programs on both topology switches and POX controllers, and a centralized scheduler program. This section briefly describes the functions of the statistics collection program and the centralized scheduler program. The “Evaluation System Architecture” section will describe these programs in details.

Statistics collection program The statistics collection program polls the statistical data from both topology switches and POX controllers. These programs are deployed locally (within the same host) to either Mininet topology or POX controller. It collects the nearly “real-time” data from both switches and controllers (by reading log constantly). By default, the statistics collection program evaluates the system every $T_p$ time interval by looking up the data collected within this period as well as those in history. From controllers, the statistics collection program assembles information such as CPU usage, memory usage, average number of events the controller handle per second, average processing time per event and number of packets dropped by the controller. From switches, the average number of flows and the packets missed will be monitored. All these statistics are stored in a centralized storage location, currently, in log files. In addition, Statistics collection programs also generate events based on controllers’ or switches-to-controller’s UP/DOWN state. As the data flow grows, storing the statistics and events could be a bottleneck, and one of our future works is to adapt these data storing tasks (disk IO) in an asynchronous logging framework, so that the disk IO tasks will not become the bottleneck in the statistics collection programs.

Centralized scheduler program The centralized scheduler program is the key application of our project, and it serves as the brain in terms of making decision on how to dynamically rebalance the controllers to switches assignment based on the “real-time” statistics feedbacks from statistics collection programs mentioned above. To determine if a controller is overloaded or under-loaded, it applies the “reassignment algorithm” from the Algorithm section, to dynamically reassign controllers to switches, so that all the topologies can perform within the users required specifications while the total number of controllers is minimized. This program runs in 2 different modes, namely, polling mode and event trigger mode. In the case of achieving failover, event triggered mode is applied to achieve fail-fast.

Evaluation and Experiment

The system design of our project requires the evaluation to be performed on multiple physical nodes. However, due to the constraints of physical resources, we have decided to simulate multiple nodes by running multiple Virtual Machines via VMWare on a PC. The PC has a CPU of Intel Core i5-3210M@2.5GHz and with 12GB RAM. Conceptually, dual cores with hyper threading allow us to run 4 Virtual Machines on the PC without much resource contention. To effectively facilitate our proof-of-concept evaluation, we decided to host our home-grown applications in 3 Virtual Machines, i.e. the PC we used is fully capable of running the evaluation. In terms of the Virtual Machine environment setup, we used default configuration provided in the Mininet’s official website, in which the mininet-2.1 and POX network OS have been installed.

1. Evaluation System Architecture

As mentioned above, our home-grown applications have been placed in 3 Virtual Machines, namely, 2 Mininet topology emulation servers and 1 scheduler application server. This section will illustrate the functions of those applications shown in Figure 1.

1.1 Mininet topology server

Mininet topology It was built using a Mininet built-in tree topology with depth=3 and fanout=3, which results in 27 hosts and 13 switches in the topology. The main function of this topology is to generate sufficient PACKET_IN events by simply executing the “pingall” command within the topology.
Stats Collector App It has three categories of functions. Firstly, it serves the functions of collecting the statistics and events from the Mininet topology running on the same Virtual Machine, such as the Mininet topology’s switches’ packet drop statistics, etc. It has a heartbeat program to continuously check if all of the switches are connected to an ACTIVE controller. In case some of the controllers are not connected to any ACTIVE controllers, it will instantly trigger a special event to the remote Scheduler App in the scheduler application server. Secondly, it serves as a data producer to centralized data storage, i.e. it sends the data collected on top of the local topology to a remote database, and the Scheduler App would use these data for decision making. Lastly, it serves as an activity executor for activities involving changes on top of its corresponding Mininet topology. For instance, if the remote Scheduler App decides to reassign half of the switches to a different remote controller, it will communicate to this Stats Collector App, and the actual command of change the controller will be executed by this Stats Collector App. In addition, the communication protocol from this Stats Collector App to the remote database or to the remote Scheduler App, is implemented in a rudimentary client-server socket programming using TCP/IP. Knowing that it might cause problems like multi-version concurrency control, or multithreading deadlock, etc., we plan to switch it to a benchmark messaging queue (e.g. ActiveMQ, RabbitMQ, etc.) in the future.

1.2 Scheduler application server

Stats Collector App Similar to the Stats Collector App in the Mininet topology server mentioned above, it serves two categories of functions. Firstly, it serves as an application of collecting the statistics and events from the host’s Mininet controllers. For instance, it polls the average PACKET_IN event processing time of each controller and collects system-level metrics for each individual controller like CPU usage, memory usage, I/O counters, connections and open files, etc. Secondly, it sends the statistics data collected on top of the controllers to a database, and the Scheduler App would use these data for deciding controller deployment.

Mininet Controllers It is built by Mininet’s built-in forwarding.l2_learning controller program running on top of POX network OS. The main reason of using forwarding.l2_learning controller is that it generates a new PACKET_IN event for every new path connection, which could consequently results in sufficient number of PACKET_IN events by simply applying Mininet’s “pingall” command on the topology from the Topology server.

Scheduler App It is the key application of our project, which serves as the brain in terms of deciding dynamical the controllers-to-switches assignment based on the “real-time” statistics feedbacks from Stats Collector App on both the Mininet topology server and the scheduler application server. To determine if a controller is overloaded or not, it applies the “evaluation function” from the Algorithm section of this report. And then it applies the “reassignment algorithm” from the Algorithm section, to dynamically rebalance the assignments between controllers to switches, so that all the topologies can perform within the users required specifications while the total number of controllers is minimized. Generally, the Scheduler App is scheduled to run in a polling mode, however, the Scheduler App also has an event trigger mode to dynamically listen to the events likes “controller UP/DOWN” event, as well as “switch-to-controller connection DOWN” event. In the case of “switch-to-controller connection DOWN” event, the Scheduler App immediately schedules a new/existing controller for those corresponding switches, which is how the failover is achieved.
2. Evaluation One: Failover

As mentioned in the evaluation system architecture, the controller failover is achieved through a heartbeat program in the Mininet topology server’s Stats Collector App, which continuously polls the connection state between the switches and an ACTIVE controller. If the connection state is down, it instantly generates an event to the remote Scheduler App in the scheduler application. Correspondingly, the Scheduler App immediately gets triggered and schedules a new/existing controller for those corresponding switches. Our initial evaluation result shows that the switches can be reconnected to an ACTIVE time controller within 3s on average. The heartbeat program in the Mininet topology server’s Stats Collector App is currently scheduled to run every 3 seconds, and this will cause problems for large-event-flow topologies. One of our future works is to maintain a mapping table of connections between controllers and switches. Therefore, a “controller DOWN” event can be instantly consumed by the Scheduler App, and the Scheduler App can run its logic to schedules a new/existing controller for those switches originally mapped to the DOWN controller. With this, it would save maximum 3 seconds for the failover to take effect.

3. Evaluation Two: Controller Over-loaded

Under the typical environment, executing a “pingall” command on top of the above-mentioned Mininet topology (27 hosts, 13 switches) does not generate sufficient loads to over-load the POX l2_learning controller. One of the problems encountered in this evaluation was to find a good way to simulate the situation when the controller faces demanding event loads (average response time to a packet-in event grows significantly, or even times out). Initially we tried to increase the controller loading by simply adding more hosts and switches to Mininet topology, however, the “pingall” command executes the “ping” between 2 hosts one by one, so the rate of PACKET_IN events is roughly the same regardless the size of the topology. We decided to look for a solution to limit the controller’s capacity of dealing with the PACKET_IN events, which would consequently speed up the controller overloading process in a flexible manner. It was found that the Linux “cpulimit” utility tool could easily limit the CPU resources of a process, and the test shows it worked fine for our requirement. (Figure 2)

![CPU-limit effect on average respond time](image)

Figure 2

Note: Different lines represent the number of hosts in Mininet topology.

Another issue encountered was that the packet processing time-out specifications of a Mininet switch could vary from topology to topology, while there is only one controller’s PACKET_IN processing time-out regardless which topology the underlying switches belong to. In order to keep the controller transparent from the various processing time-out specifications of its underlying switches, our solution detects the time-out events from the perspective of switches from individual topology. Our solution exploiting the built-in utility tool “ovs-dpctl” from Mininet since all of our evaluation was performed on top of Mininet Open vSwitch (OVS) [2]. The “ovs-dpctl” utility’s return has a field named “hit”, which indicates the number of packet processing time-out event occurs on the OVS level. In our solution, there is a dedicated script (scheduled to run every 3 seconds) deployed in the Stats Collector App in the Mininet topology server, which keeps track of the “hit” number from “ovs-dpctl” return, and it will send a “Controller-Overloaded” trigger event to the Scheduler App to run the reassignment algorithm when the “hit” number increases. This is the event-trigger mode of running the reassignment algorithm.

In this evaluation, another indicator of “Controller-Overloaded” situation is that the controller’s average PACKET_IN event processing time has increased by 50% over the previous snapshot. To achieve this, the Scheduler App is scheduled every 15 seconds to check the average PACKET_IN event processing time collected by the
**Stats Collector App** in the Scheduler server. If it detects an increment of 50% over the previous snapshot, it will run the reassignment algorithm. This is the polling mode of running the reassignment algorithm. Figure 3 illustrates these two modes.

![Figure 3](image)

Our initial evaluation result shows that the switches can be reassigned to a normal loaded controller within 8s on average in the polling mode, while this can be completed within 3s on average in the event-trigger mode.

**Discussion**

During the project, we found the following questions that deserve further discussion:

1. How to simulate a huge network topology with demanding traffics close to the real world?

Advanced SDN tools such as Mininet, OVS provide us convenience in SDN framework simulation. However, simulating large networks via Mininet with limited resources is not easy. At the beginning, we tried to deploy Mininet topology and multiple POX controllers on the same virtual machine (VM), but sharing computing resources among Mininet topology and controllers degrades the performance of controllers, especially when the network size grows. This problem can be solved if the CPU and memory resources of Mininet could be isolated from POX controllers. Later on, we tried running Mininet and POX controllers in separate bridged VMs. The loading of the simulated network could be easily adjusted by simply modifying the number of bridged VMs, which increased the flexibility of the simulation. In order to simulate the bottleneck of controllers’ capacity, we adopted “cpulimit” utility tool to restrict the distributed CPU resources for each controller. It will be more convenient if such CPU usage limiter can become a built-in feature of SDN tools in the future.

2. What visualization-tools can be used in demonstration?

There are some visualization-tools now suitable for Mininet and different controllers. However, to the best of our knowledge, most of them don’t support dynamic network topology, which is a significant issue in SDN research. Moreover, they are not compatible to the experiment scenarios we expected. A more flexible and user-friendly visualization-tool are preferred in system demonstration and data analysis.

**Summary and Future Works**

For this project, we have designed the basic evaluation function of controller performance and resources reallocation algorithm for controllers’ dynamic placement. In framework evaluation, we designed a communication protocol between topology server and scheduler application server based on TCP/IP. We evaluated our proposed system in two experiments based on Mininet topology and POX controllers, with the help of the home-grown statistics collection and centralized scheduler applications. In the experiments, the applications successfully reassigned the controllers to different switches and prevented packets miss according to real time statistics feedbacks.

There are some possible future works to do, such as considering more metrics in the centralized scheduler program, and improving reassignment algorithm for better load balancing. Also, we plan to adapt the collected data storing tasks (disk IO) in an asynchronous logging framework, so as to avoid disk IO tasks bottleneck in the statistics collection programs. In order to prevent problems like multi-version concurrency control or multithreading deadlock, we will modify the communication protocol among applications to a benchmark messaging queue in the future. Furthermore, we suggest maintaining a mapping table of controllers-and-switches association, aiming to accelerate failover detection process.
Task Division

Sherrill:
1. Investigate on related works (dynamic controller schedulers, dynamic resources allocation algorithm).
2. Design the evaluation function and reassignment algorithm.
3. Implement the topology used for testing.

Marc:
1. Investigate on the available library in allocation scheduler.
2. Survey and select the metrics (CPU utilization, events per sec) to consider in the scheduler.
3. Coordinate the group meeting time and task division.

Dominic:
1. Implement the real-time metrics feedback collector based on Pox.
2. Implement the basic functional units for the scheduler (switch-to-controller reconnection, statistics collection).
3. Design the experiment.

References