# Dynamic Controller Provisioning in Software Defined Networks

Md. Faizul Bari, Arup Raton Roy, Shihabur Rahman Chowdhury, Qi Zhang,
Mohamed Faten Zhani, Reaz Ahmed, and Raouf Boutaba
David R. Cheriton School of Computer Science,University of Waterloo
Email: {mfbari, ar3roy, sr2chowdhury, q8zhang, mfzhani, r5ahmed, rboutaba}@uwaterloo.ca

***Abstract*—Software Defined Networking (SDN) has emerged as a new paradigm that offers the programmability required to dynamically configure and control a network. A traditional SDN implementation relies on a logically centralized controller that runs the control plane. However, in a large-scale WAN deployment, this rudimentary centralized approach has several limitations related to performance and scalability. To address these issues, recent proposals have advocated deploying multiple controllers that work cooperatively to control a network. Nonetheless, this approach drags in an interesting problem, which we call the Dynamic Controller Provisioning Problem (DCPP). DCPP dynamically adapts the number of controllers and their locations with changing network conditions, in order to minimize flow setup time and communication overhead. In this paper, we propose a framework for deploying multiple controllers within an WAN. Our framework dynamically adjusts the number of active controllers and delegates each controller with a subset of Openflow switches according to network dynamics while ensuring minimal flow setup time and communication overhead. To this end, we formulate the optimal controller provisioning problem as an Integer Linear Program (ILP) and propose two heuristics to solve it. Simulation results show that our solution minimizes flow setup time while incurring very low communication overhead.**

## I. Introduction

Software defined networking (SDN) has emerged as a new paradigm that offers the programmability required to dynamically configure and manage the network. By separating the control plane from the data plane and shifting the control plane to a conceptually centralized controller, SDN provides network operators with a strong capability to implement a wide-range of network policies (*e.g.*, routing, security, fault-tolerance) and the ability to rapidly deploy new network technologies.

The most common SDN implementation today relies on a logically centralized controller that possesses a global view of the network. Whenever a switch receives a new flow, it requests the controller to install appropriate forwarding rules along the desired flow path. The time required to complete this operation is known as the flow setup time. However, in a large-scale WAN deployment, this rudimentary centralized approach has several limitations related to performance and scalability. First, it is not always possible to find an optimal placement of the controller that can ensure acceptable latencies between the controller and the switches situated at geographically distributed locations. Secondly, a single controller usually has a limited resource capacity and hence cannot handle large amount of flows originating from all the infrastructure switches. In this case, the average flow setup time can rise significantly and degrade application and service performance [15].

To address these limitations, recent proposals have advocated deploying multiple controllers that work cooperatively to better manage network traffic flows [7], [14]. Nonetheless, this approach introduces a new problem: minimizing flow setup times by dynamically adapting the number of controllers and their locations according to demand fluctuations in the network. We call this problem the Dynamic Controller Provisioning Problem (DCPP). Specifically, DCPP requires the number of controllers to be sufficient to handle the current network traffic, and their locations should ensure low switch-to-controller latencies. However, the multi-controller deployment also requires regular state synchronization between the controllers to maintain a consistent view of the network [12]. This communication overhead can be significant if the number of controllers in the network is large. Finally, as network traffic patterns and volumes at different locations can vary significantly over-time, the controller placement scheme has to react to network "hotspots" and dynamically re-adjust the number and the location of controllers. Hence, the solution to DCPP must always find the right trade-off between performance (in terms of flow setup time) and overhead (controller synchronization and management).

To the best of our knowledge, the only work that has investigated the controller placement problem is the one by Heller et al. [8]. They studied a static version of the problem where controller placement remain fixed over time, and analyzed the impact of the controller locations on the average and worst-case controller-to-switch propagation delay. However, a static controller placement configuration may not be suitable forever as network conditions can change over time.

To address this limitation, we propose a management framework for dynamically deploying multiple controllers within an WAN (Section III). Specifically, we consider the dynamic version of the controller placement problem where both the numbers and locations of controllers are adjusted according to network dynamics. Our solution takes into account the dynamics of traffic patterns in the network, while minimizing costs for (i) switch state collection, (ii) inter-controller synchronization, and (iii) switch-to-controller reassignment. We formulate DCPP as an Integer Linear Program (ILP) that considers all aforementioned costs (Section IV). We then propose two heuristics that dynamically estimates the number

ISBN 978-3-901882-53-1, 9th CNSM and Workshops ©2013 IFIP 

of controllers and decide their placement in order to achieve the desired objectives (Section V). The effectiveness of our solution is then demonstrated using real-world traces and WAN topologies (Section VI). Our results show that the proposed algorithms minimize the average flow setup time while incurring very low communication overhead. Finally, we provide concluding remarks (Section VII).

## II. Related Work

SDN aims at decoupling the control plane from the data plane. However, the original SDN architecture was designed to use a centralized control plane, which is known to have poor scalability for managing large networks. Recent research works have proposed a number of techniques to overcome this scalability limitation. 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 controllers.

DevoFlow [5] and DIFANE [16] falls in the first category of techniques. DevoFlow proposes to pre-install wildcard rules in the switches that can replicate themselves for the mice flows to create flow specific rules. The switches also have intelligence to detect elephant flows. The controller is only responsible for making forwarding decision for elephant flows. Similarly, in DIFANE, the controller generates the forwarding rules, but it is not involved in the setup of each new flow. Rather, the rules are partitioned and distributed among a subset of switches called "authoritative switches". The regular switches, which forward packets in data plane, redirect new flows to the authoritative switches to learn about the forwarding rules. However, both of these proposals require some changes to be made to the commodity switches to increase their intelligence.

On the other hand, Kandoo [7], HyperFlow [14], and Onix [10] propose to distribute the control plane across multiple controllers to improve SDN's scalability. Each of them distributes controller states differently. Kandoo distributes controller states by placing the controllers in a two level hierarchy comprising a root controller and multiple local controllers. Local controllers respond to the events that do not depend on global network state (*e.g.*, elephant flow detection), while the root controller takes actions that require global network view (*e.g.*, re-routing elephant flows). HyperFlow handles state distribution of the distributed controllers through a publish/subscribe system based on the WheelFS distributed file system. Finally, controller state distribution in Onix is managed through a distributed hash table.

However, none of the aforementioned works consider the issue of choosing suitable network locations for placing controllers and adapting the placement according to the dynamic behavior of the network. The problem regarding how many controllers to place and where to place them in the network was first studied by Heller *et al.* [8]. They analyzed the impact of placing multiple controllers according to different heuristics in a static setting and did not consider adapting the number and placement of controllers with changing traffic load. On the contrary, we propose a management framework that takes both network performance (in terms of flow setup time) and management overhead (for state synchronization) into consideration to determine the number and placement of controllers in the network. Furthermore, we aim at dynamically provisioning SDN controllers over time to react with traffic fluctuations.

## III. System Description

In this work, we consider a large WAN consisting of OpenFlow enabled switches to deploy our system. However, our proposed system also works with WANs with a mix of OpenFlow and non-OpenFlow switches, where non-OpenFlow switch simply work as forwarding elements. We also assume that the network operator has provision to deploy or has already deployed compute resources (*e.g.*, servers) at particular locations throughout the network. These servers are used to deploy SDN controllers to control the OpenFlow enabled switches in the network.

As explained in Section I, a single controller is not sufficient for large WAN deployments. Hence, our system dynamically partitions the set of OpenFlow switches into multiple *domains* (henceforth we use "domain" to specify the set of OpenFlow switches that are controller by a OpenFlow controller) based on network dynamics and assigns one controller per domain. At any time instance, a switch is controlled by a single controller and each controller is responsible for setting up paths in switches under its own domain.

A controller periodically collects port, flow, and table level statistics from switches in its domain using `OFPST_PORT`, `OFPST_FLOW`, and `OFPST_TABLE` OpenFlow messages, respectively. Controllers exchange switch and link level information with one another so that each controller can take forwarding decisions. The protocol for inter-controller communication is out-of-scope for this paper and we plan to explore it in the future. Each controller builds its own view of the network from the exchanged information.

In our system controllers are always running on servers located at designated network locations. A controller is regarded as *active* if it has at least one switch assigned to it, otherwise it is considered *inactive*. Inactive controllers keep listening on a particular port for incoming `HELLO` messages from newly assigned switches and consume very small amount of CPU cycles and power. Our management framework periodically evaluates the current switch-to-controller assignment and decides whether to perform a reassignment based on the specified constraints. If a reassignment is performed, the management framework also changes the switch-to-controller assignment in the network. Our management framework contains three modules as depicted in Fig. 1 and explained below:

- *Monitoring Module* monitors controllers through periodic heartbeat messages to ensure aliveness and pulls relevant statistics from them.
- *Reassignment Module* periodically checks the collected statistics by the monitoring module and decides whether to perform a reassignment. This decision depends on many criteria that will be explained in-detail in the

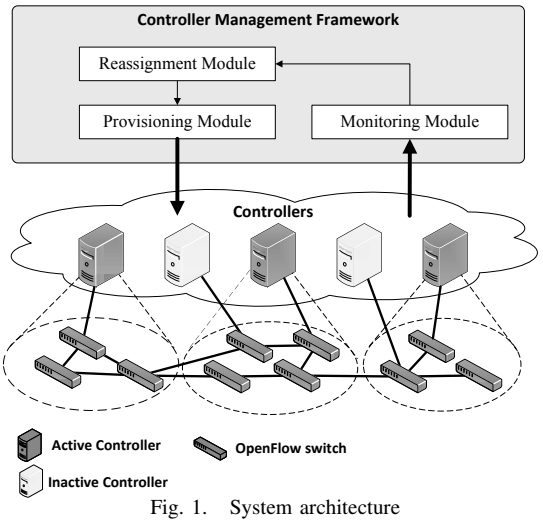



Fig. 1. System architecture

following section. This module is also triggered by the monitoring module to perform an instant reassignment in case of a controller failure.

- *Provisioning Module* provisions controllers as required and changes switch to controller associations.

During a reassignment an active controller may become inactive, if all of its switches are assigned to other controllers and an inactive controller may become active, if any switch is assigned to it. The objective of our system is to keep a set of controllers in the active state that results in close to optimal flow setup times, while incurring low communication overhead.

In our system, when a new flow arrives at a switch, the switch first checks its forwarding table for a matching entry. If a matching forwarding rule exists, packets in the flow are forwarded according to the matching rule. If no such rule exists, the switch sends a `PACKET_IN` OpenFlow message to its controller. The controller computes a path contained within its domain using its local knowledge about the network and installs forwarding rules in the switches under its domain. Two possible cases can arise next: (i) the flow may pass only through switches under the current controller's domain, or (ii) it may pass though one or more other domains. In the first case, we are already done as all switches in the flow's path have the necessary forwarding rules installed in their forwarding tables. In the second case, additional flow setup requests will be generated as shown in Fig. 2. In this figure, a new flow arrives at switch $i$ on port $a$. As the switch does not have a matching forwarding entry, it sends a flow setup request to its controller $m$. This request is called *Initial path setup request*. Now, controller $m$ computes a path (contained within its own domain) for the flow. Lets say the path is $i.a \rightarrow i.b \rightarrow j.b \rightarrow j.c$ (where $i.a$ means port $a$ of switch $i$) and sets up the forwarding rules in switches $i$ and $j$. The packets in the flow are forwarded from port $a$ to port $b$ of switch $i$ and then from port $b$ to port $c$ of switch $j$, eventually reaching port $c$ of switch $k$. Switch $k$ now searches its forwarding table for a matching rule. If no such rule exists, it sends a flow setup request to its controller $n$. This request

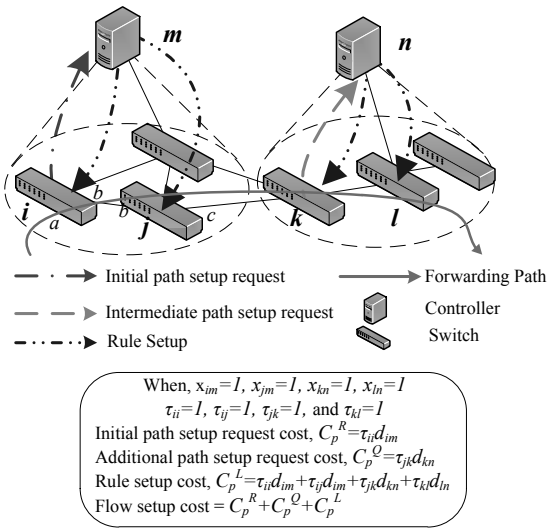



Fig. 2. Path setup method with Flow setup cost

is termed as *Intermediate path setup request*. Now, controller $n$ computes a path contained in its own domain and installs forwarding rules in switch $k$ and $l$ in a similar manner.

## IV. Problem Formulation

In this section, we formulate DCPP as an ILP. Specifically, we model the network as an undirected graph, $G = (S, E)$, where $S$ is the set of switches and $E$ is the set of edges. Let $d_{ij}$ denote the cost of the shortest path between switches $i$ and $j$ expressed in terms of propagation delay or number of hops. $F$ ($F \subseteq S$) is the set of locations where a controller can be deployed. Let vector $U = \langle u_1, u_2, \ldots u_{|F|} \rangle$ represents the capacities of the controllers. Hence, $u_m$ is the maximum number of requests controller $m$ can handle per second. The maximum allowable cost between a switch and its controller is denoted by $\delta$ (expressed in the same unit as $d_{ij}$).

The traffic matrix is denoted by $\mathcal{T} = [\tau_{ij}]_{|S|\times|S|}$, where $\tau_{ij}$ represents the average number of flows over the current time slot originating from switch $i$ to its neighbor switch $j$ as reported by the monitoring module. Moreover, each diagonal entry $\tau_{ii}$ of $\mathcal{T}$ captures the average number of flows originated at switch $i$, *i.e.*, the average number of flows coming from the networks served by switch $i$.

The reassignment algorithm is invoked at every $T_a$ time interval. The output of our ILP is an assignment matrix $\mathcal{X} = [x_{im}]_{|S|\times|F|}$, where $x_{im}$ is equal to 1 if switch $i$ is assigned to controller $m$, and 0 otherwise. It also provides a binary vector $Y = \langle y_1, y_2, \ldots y_{|F|} \rangle$ indicating which controllers are active (*i.e.*, $y_m = 1$) and which are not (*i.e.*, $y_m = 0$).

We consider the following four costs that will be incurred when deploying multiple controllers across the network:

1) **Statistics collection cost** ($C_l$) is the number of messages per second required for the controllers to collect statistics from their associated switches. Assuming that statistics are gathered at each time interval $T_s$ (note that $T_s < T_a$), this cost can be expressed as follows:

$$C_l = \left\lfloor \frac{T_a}{T_s} \right\rfloor \sum_{i \in S} \sum_{m \in F} d_{im} x_{im} \qquad (1)$$

2) **Flow setup cost** ($C_p$) is the total cost incurred for setting up the flow rules across end-to-end paths. As explained in Fig. 2, this cost can be divided into three components. First, the initial path setup request for the flows originated at the switches:

$$C_p{}^R = \sum_{i \in S} \sum_{m \in F} \tau_{ii} x_{im} d_{im} \tag{2}$$

Secondly, the intermediate path setup requests at each switch for the flows coming from a neighbor switch controlled by a different controller:

$$C_p{}^Q = \sum_{i \in S} \sum_{j \in S} \sum_{m \in F} \sum_{n \in F} \tau_{ji} x_{jn} (1 - x_{in}) x_{im} d_{im} \tag{3}$$

Finally, the rule installation cost incurred for the rule installation messages from the controllers:

$$C_p{}^L = \sum_{i \in S} \sum_{j \in S} \sum_{m \in F} \tau_{ji} x_{im} d_{im} \tag{4}$$

Combining Equations (2), (3), and (4), we can derive the flow setup cost as follows:

$$C_p = C_p{}^R + C_p{}^Q + C_p{}^L \tag{5}$$

3) **Synchronization cost** ($C_s$) represents the number of messages exchanged between controllers in order to maintain a consistent network-wide view in all of them. We assume messages are exchanged every $T_x$ seconds (note that $T_x < T_a$). We also consider critical events that force a controller to instantaneously synchronize state with other controllers. Assuming $e$ is a random variable that represents the occurrence frequency of critical events in the system. We can define the number of inter-controller state synchronization messages generated within time $T_a$ considering both periodic and critical events as follows:

$$N_{\mathcal{E}} = \left\lfloor \frac{T_a}{T_x} \right\rfloor + \int_0^{T_a} e \cdot p(e) de \tag{6}$$

Here, $p(e)$ is the probability distribution function of $e$. The synchronization cost can be defined as follows:

$$C_s = \frac{N_{\mathcal{E}}}{T_a} \sum_{m \in F} \sum_{n \in F} y_m y_n d_{mn} \tag{7}$$

4) **Switch reassignment cost** ($C_r$) is the cost of assigning a switch to a new controller. Ideally, it is better to avoid frequent reassignment of switches. Assume that the previous assignment is given by the matrix $\tilde{\mathcal{X}} = [\tilde{x}_{im}]_{|S| \times |F|}$. We define the matrix $\mathcal{Z} = [z_{im}]_{|S| \times |F|}$ as the XOR between the new assignment $\mathcal{X}$ and the previous assignment $\tilde{\mathcal{X}}$. In particular, $z_{im} = 1$ if the assignment of switch $i$ has been changed to (or from) controller $m$, otherwise $z_{im} = 0$.

$$C_r = \sum_{i \in S} \sum_{m \in F} d_{im} z_{im} \tag{8}$$

The objective of our optimization problem is to minimize the weighted sum of the aforementioned mentioned four costs and can be expressed as follows:

$$\alpha C_l + \beta C_p + \gamma C_s + \lambda C_r \tag{9}$$

Here, $\alpha$, $\beta$, $\gamma$, and $\lambda$ are constants, the network operator can use to adjust the relative significance of the four cost components. Furthermore, the following constraints must be satisfied in order to guarantee a feasible solution:

$$\forall_{i \in S}: \sum_{m \in F} x_{im} = 1 \tag{10}$$

$$\forall_{m \in F}: \sum_{i \in S} x_{im} \tau_{ii} + \sum_{i \in S} \sum_{j \in S} \sum_{n \in F} \tau_{ji} x_{jn} (1 - x_{in}) x_{im} \leq y_m u_m \tag{11}$$

$$\forall_{i \in S, m \in F}: x_{im} d_{im} \leq \delta \tag{12}$$

$$\forall_{i \in S, m \in F}: x_{im} \leq y_m \tag{13}$$

$$\begin{aligned} \forall_{i \in S, m \in F}: \; & z_{im} \leq x_{im} + \tilde{x}_{im} \\ & z_{im} \geq x_{im} - \tilde{x}_{im} \\ & z_{im} \geq -x_{im} + \tilde{x}_{im} \\ & z_{im} \leq 2 - x_{im} - \tilde{x}_{im} \end{aligned} \tag{14}$$

$$\forall_{i \in S, m \in F}: x_{im}, z_{im} \in \{0, 1\} \tag{15}$$

$$\forall_{m \in F}: y_m \in \{0, 1\} \tag{16}$$

Constraint (10) guarantees that every switch is controlled by exactly one controller at a given time. Inequality (11) ensures that a controller can satisfy the path setup requests from the switches assigned to it. Note that the total number of path setup requests to a controller is composed of all initial and intermediate path setup requests from all switches that it is currently controlling. Inequality (12) gives an upper bound $\delta$ on the maximum delay between a switch and its designated controller. The condition on assigning a switch to an active controller is represented by Inequality (13). The inequalities of (14) ensure that $z_{im}$ is the XOR of the variables $x_{im}$ and $\tilde{x}_{im}$. Equations (15) and (16) indicate that $x_{im}, y_m$, and $z_{im}$ are binary variables. This formulation generalizes the *Single Source Unsplittable Flow Problem* [9], which is known to be $\mathcal{NP}$-Hard. Therefore, we propose two heuristics to solve this problem that are described in the subsequent sections.

## V. Proposed Heuristic

In this section, we describe two heuristics for solving DCPP: (i) *DCP-GK*: a greedy approach based on the knapsack problem, and (ii) *DCP-SA*: a simulated annealing based meta-heuristic approach. The input to both heuristics include network topology $G$, traffic matrix $\mathcal{T}$, previous switch–to–controller assignment $\tilde{\mathcal{X}}$, set of switches $S$, possible controller locations $F$, controller capacity vector $U$, and delay constraint $\delta$. The goal of these heuristics is to find a feasible switch–to–controller assignment that minimizes the cost function expressed in Equation (9) based on current network conditions.

### A. Dynamic Controller Provisioning with Greedy Knapsack (DCP-GK)

Here, we model each controller as a knapsack. The capacity of each knapsack is equal to the processing capacity (measured in number of flow setup requests it can handle per time interval, 60 minutes in our simulations) of its corresponding controller. We consider the switch as the objects to be added in the knapsack. We model the weight of a switch as the number of new flows it generates within the previous time interval and the profit of taking a switch is the inverse path cost between the switch and that controller. Each iteration of our algorithm activates a single controller. This controller is chosen such that the sum of path costs from that controller to the unassigned switches is minimum and within the given delay bound $\delta$. Then we run the greedy knapsack algorithm to assign switches to that controller. If no switch could be assigned to a controller it is deactivated. The iterations stop when all the switches are assigned to a controller or no more controllers can be activated. If there are unassigned switches after all the iterations are completed, the switches are assigned randomly between the activated controllers. This may break the capacity and delay constraints. However, this exceptional case occurred very rarely during our simulations.

### B. Dynamic Controller Provisioning with Simulated Annealing (DCP-SA)

The DCP-SA heuristic provides a feasible switch-to-controller assignment $\mathcal{X}$ considering the previous assignment matrix $\tilde{\mathcal{X}}$ as an initial state for simulated annealing. However, due to a change in traffic pattern, $\tilde{\mathcal{X}}$ may violate the capacity constraint depicted in Equation (11). Therefore, the objective of Algorithm 1 is to generate a feasible switch to controller assignment from the current unfeasible assignment. The output of this algorithm is provided as an input to Algorithm 2 which runs the simulated annealing algorithm to improve the switch-to-controller assignment.

More specifically, Algorithm 1 first identifies the set of controllers $F_v$ for which capacity constraints are violated. Then it tries to lower the load on each $f \in F_v$ by reassigning one or more switches to other controllers without violating the capacity constraint. To achieve this objective, Algorithm 1 first sorts all switches $S_f$ assigned to controller $f$ according to their rank defined by the following equation:

$$r_i = \sum_{j \in S} \tau_{ij} \tag{17}$$

Let $s^*$ denote the switch with the highest rank in $S_f$. A set of feasible controllers $F_{s^*}$ is identified for $s^*$ such that each controller in $F_{s^*}$ is within the bound $\delta$ from $s^*$ and also has sufficient capacity to handle requests from $s^*$. The algorithm then selects the controller with the smallest remaining capacity $\tilde{f}$, and assigns $s^*$ to $\tilde{f}$. The intuition is to minimize the fragmentation of remaining capacity of the controllers during the reallocation. The assignment matrix $\tilde{\mathcal{X}}$ is then updated accordingly. This reallocation procedure for controller $f$ continues until the capacity constraint for $f$ is satisfied. Algorithm 1 repeats this reallocation procedure for each of the controllers in set $F_v$ until the switch to controller assignment $\tilde{\mathcal{X}}$ becomes feasible.

**Algorithm 1** Algorithm for generating feasible initial state

**Input:** Topology, $G$
Traffic Matrix, $\mathcal{T}$
Previous Assignment, $\tilde{\mathcal{X}}$
Set of switches, $S$
Set of controllers, $F$
Controller capacity vector, $U$
**Output:** New Feasible Assignment, $\tilde{\mathcal{X}}$

$F_v \leftarrow$ Set of controllers for which $\tilde{\mathcal{X}}$ violates capacity constraints
**while** $F_v \neq \emptyset$ **do**
  Select a controller $f$ from $F_v$
  $S_f \leftarrow$ Set of switches assigned to controller $f$
  **while** Capacity of $f$ is violated **do**
    Sort $S_f$ according $r_i$ defined by Equation (17)
    $s^* \leftarrow$ first node in $S_f$
    $F_{s^*} \leftarrow$ Feasible controllers of $s^*$ with remaining capacity greater than the demand of $s^*$
    **if** $F_{s^*} \neq \emptyset$ **then**
      $\tilde{f} \leftarrow$ Controller with smallest remaining capacity in $F_{s^*}$
      Assign $s^*$ from $f$ to $\tilde{f}$ and update $\tilde{\mathcal{X}}$
    **end if**
    $S_f \leftarrow S_f \setminus \{s^*\}$
  **end while**
  $F_v \leftarrow F_v \setminus \{f\}$
**end while**

**Algorithm 2** Reassignment algorithm

**Input:** Topology, $G$
Traffic Matrix, $\mathcal{T}$
Feasible Previous Assignment, $\tilde{\mathcal{X}}$
Set of switches, $S$
Set of controllers, $F$
Controller capacity vector, $U$
**Output:** New Assignment, $\mathcal{X}$

$\mathcal{X} \leftarrow \tilde{\mathcal{X}}$
$F_S \leftarrow$ Feasible controllers for $S$ considering delay constraints
Select an initial temperature $Temp > 0$
$current \leftarrow \tilde{\mathcal{X}}$
**for** $t \leftarrow 1$ to $\infty$ **do**
  $Temp \leftarrow$**Schedule**$(t)$
  **if** $Temp = 0$ **then**
    **break**
  **end if**
  $i \leftarrow 1$
  **repeat**
    $next \leftarrow$ **Successor**$(current, F_S)$
    $\Delta \leftarrow$ **Cost**$(current) -$ **Cost**$(next)$
    **if** $\Delta > 0$ **then**
      $current \leftarrow next$
    **else**
      $current \leftarrow next$ only with $e^{\frac{\Delta}{Temp}}$ probability
    **end if**
    **if** **Cost**$(current) <$ **Cost**$(\mathcal{X})$ **then**
      $\mathcal{X} \leftarrow current$
    **end if**
    $i \leftarrow i + 1$
  **until** $i \neq N$
**end for**

Starting from a feasible assignment $\tilde{\mathcal{X}}$, Algorithm 2 uses a variant of simulated annealing to further optimize the assignment. We define the following local search moves for this algorithm (sorted in decreasing order of preference):

- **Relocate Switch**: selects a switch randomly and assigns it to a different active controller. If no switch is assigned to a controller after this move, it is deactivated.
- **Swap switches**: selects two switches randomly from two

different controllers and swap their assignments.

- **Activate controller**: activates a randomly chosen inactive controller.
- **Merge assignments**: randomly selects two controllers and reassigns all switches of one controller to the other. The idle controller is then deactivated.

The *Successor* procedure in Algorithm 2 returns a random next state from the current state using one of the aforementioned moves. This procedure always returns a feasible successor such that no constraint is violated.

## VI. Evaluation

We evaluate the performance of our proposed framework through extensive simulations. We tried different simulation methodologies to find a suitable one for our purpose. First, we tried to use Mininet [11] with POX [1] controllers deployed on the same physical machine, but found that mininet is inadequate for our purpose as it cannot handle the amount of traffic we wanted to simulate. Mininet simulates hosts and switches in separate network namespaces and connects them with virtual Ethernet interfaces. Traffic from switch-to-host, switch-to-switch, and switch-to-controller flows through the loopback interface of the physical machine. As a result, the switching capability of this loopback interface limits the amount of traffic that Mininet can simulate. As we are simulating large WAN networks, the amount of traffic is huge and the loopback interface was not able to process it in a timely manner. Second, we ran Mininet in one physical machine and ran the controllers in multiple physical machines, hoping to decrease the load on the loopback interface. However, in this case traffic from different switches were serialized through the loopback interface, whereas they should be forwarded to their respective controllers in parallel. Due to this serialization, the impact of traffic is skewed and the obtained results are characterized by the switching capability of the loopback interface, instead of the generated traffic. So, we opted for an in-house simulator where we simulate the propagation delays between switch-to-switch, switch-to-host, and switch-to-controller. Controller capacity is simulated using the results provided by Tootoonchian *et al.* [15] for the NOX [2] controller. All our simulations were conducted on a single machine with dual quad-core 2.4GHz Intel Xeon E5620 processors and 12-GB of RAM. In the following, we first describe in detail the simulation setup and the dataset we used. We then describe the metrics used to evaluate the effectiveness of our proposed framework. Finally, we compare our DCP algorithms (DCP-GK and DCP-SA) with two static scenarios: in the first case, a single controller is used for the entire network (1-CRTL), while in the second, one controller is used for each switch (N-CTRL).

### A. Simulation Setup

In our experiments, we simulate two different ISP topologies RF-I (79 nodes, 294 links) and RF-II (108 nodes, 306 links) with inter-node latencies obtained from the RocketFuel repository [13]. We assume each node in the RocketFuel topology to be an OpenFlow switch. We assume that controllers can be dynamically provisioned at any of these switches' locations. Controllers communicate with each other to exchange and synchronize switch status and port level status. Each controller computes a path for a new incoming flow from the information it has about the network in its local database and sets up paths according to the method described in Section III.

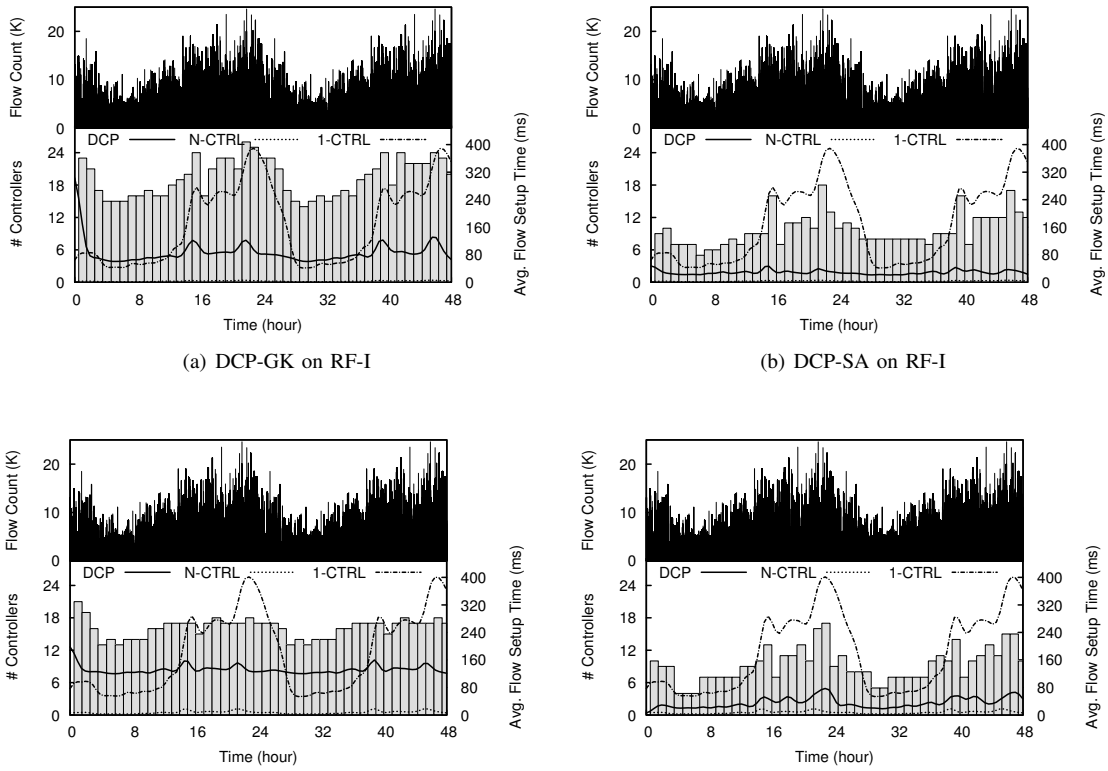
We use *iperf* to generate TCP flows between the end hosts [3]. The end hosts of each flow is chosen randomly. To make the traces more realistic, we generated the flows according to the distribution of flow sizes, flow inter arrival times, and number of concurrent flows reported in a recent study on network traffic characterization [6]. The generated traffic spans 48 hours capturing the time-of-day effect.

Finally, the management framework is implemented in Python. Specifically, the monitoring module periodically pulls statistics from the controllers, the reassignment module runs a heuristic (either DCP-GK or DCP-SA) using these statistics to find the next switch-to-controller assignment, and the provisioning module assigns switches to their controllers according to the assignment generated by the reassignment module.

### B. Results

We run each simulation for 48 hours with the reassignment heuristic running every 60 minutes. The reassignment interval can be further tuned through a more detailed analysis of the traffic. At each interval, we compute the average flow setup time, the set of active controllers, and the number of exchanged messages between the active controllers.

Fig. 3 shows the number of active controllers and average flow setup time during each interval for both topologies. For the one controller (1-CTRL) case, flow setup time varies with traffic load. If there is a peak in traffic then flow setup time also increases. A single controller cannot keep the flow setup time consistent or within acceptable limits, which is reported to be 200ms in [4] for mesh restoration. Hence, a single controller cannot provide any service guarantees. On the other hand in the N-CTRL case, the flow setup time is almost zero as expected. However, in this case the messaging overhead is much higher (as shown in Fig. 5 and explained later in this section). Fig. 3(a) and Fig. 3(b) report the above mentioned metrics for topology RF-I, using greedy knapsack (DCP-GK) and simulated annealing (DCP-SA) heuristics, respectively. DCP-GK keeps the flow setup time within 140ms and manages to keep it consistent even during traffic spikes. Even though we can see small spikes in flow setup time, none of them is as high as for the 1-CTRL case. DCP-GK also uses much fewer controllers than N-CTRL. The maximum number of controllers used by DCP-GK is only 25, during time interval 22, which is around 30% of the N-CTRL (79 controllers) case. From Fig. 3(b), we can see that DCP-SA performs much better than DCP-GK and 1-CTRL. Flow setup time is within 60ms and number of controllers is less than that of both DCP-GK and 1-CTRL. It uses a maximum of 18 controllers that is around only 23% of the N-CTRL case. Effect of traffic




(a) DCP-GK on RF-I

(b) DCP-SA on RF-I

(c) DCP-GK on RF-II

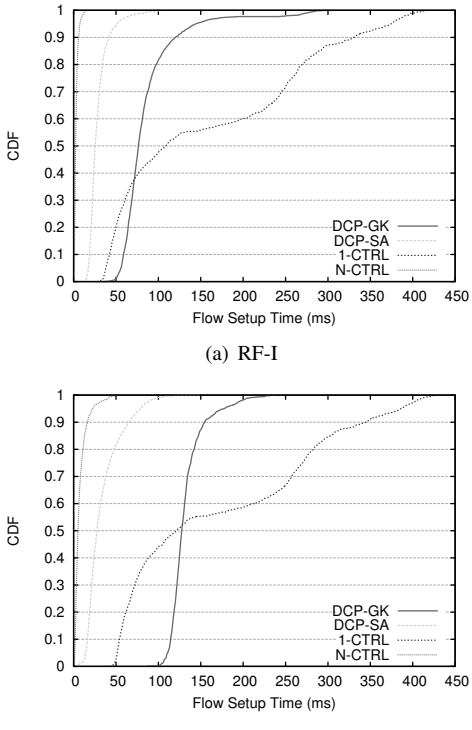
(d) DCP-SA on RF-II

Fig. 3. Controller Count and Flow Setup Time vs. Time

spikes is further reduced in this case and the flow setup time is almost constant throughout 48 hours of simulated time. This shows the effectiveness of our dynamic controller provisioning mechanism. Similar behavior is also observed for RF-II as reported in Fig. 3(c) and 3(d).

Although DCP-SA outperforms DCP-GK in both number of controllers and flow setup time, there is a penalty. DCP-SA requires much longer time to run than DCP-GK. In our simulation setup, DCP-SA took around 2 minutes to perform one reassignment for topology RF-I, whereas DCP-GK took only 0.41 seconds. For topology RF-II DCP-SA took around 4 minutes and DCP-GK took only 0.44 seconds. So, DCP-SA is preferable for near-optimal solutions and DCP-GK is suitable when we need solutions within a small amount of time.

Fig. 4(a) and Fig. 4(b) show the CDFs of flow setup time for topologies RF-I and RF-II, respectively. We can see that DCP-SA outperforms both DCP-GK and 1-CTRL in both cases by a large margin. DCP-SA always provides shorter flow setup time than 1-CTRL and all flows are setup within the acceptable range of 200ms. For RF-I, DCP-SA takes at most 120ms and for RF-II, it takes at most 150ms. While DCP-GK takes longer, it completes 99% flow setups within the acceptable range of 200ms for both topologies. On the other hand 1-CTRL can complete only 60% flow setups within 200ms and the maximum time it takes is close to 450ms (more than twice of the acceptable range) for both topologies. DCP-GK cannot outperform 1-CTRL for low flow setup time, which is evident



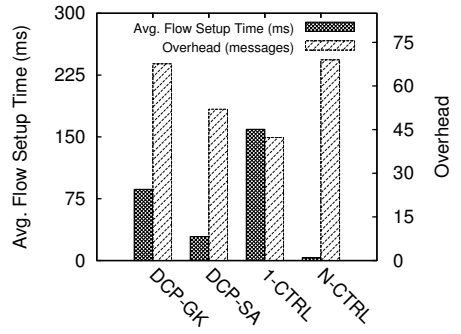

(a) RF-I

(b) RF-II

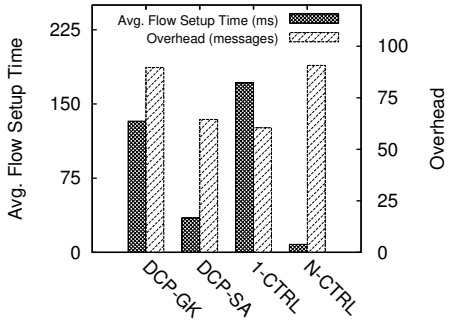
Fig. 4. CDF of Flow Setup Time

in Fig. 3(a) and Fig. 3(c). This happens during low traffic load conditions. This is a direct consequence of the greediness of

this heuristic, as it chooses the best controller at each stage without looking ahead and thereby missing a better solution. The N-CTRL case shows the lowest flow setup time, but this is a hypothetical, un-realistic lower bound. Clearly, *connecting one controller per switch* – is not an acceptable solution for this problem. We included it for comparing with the absolute lowest possible flow setup time.




(a) RF-I




(b) RF-II

Fig. 5. Summary of Overhead and Average Flow Setup Time

Fig. 5(a) and Fig. 5(b) report the messaging overhead and average flow setup time for both topologies. 1-CTRL has the lowest messaging overhead as there is no synchronization and controller-to-controller communication overhead. On the other hand, N-CTRL has the highest messaging overhead as every controller is communicating with every other controller. Messaging overhead for DCP-GK and DCP-SA is in between these two. Overhead for DCP-SA is smaller than DCP-GK as it uses fewer number of controllers and is very close to the 1-CTRL (lower bound) case for both topologies. As mentioned earlier, average flow setup time for DCP-SA is lower than both DCP-GK and 1-CTRL, but higher than N-CTRL. For topology RF-I and RF-II, average flow setup time for DCP-SA is 29 and 34ms, respectively. For the N-CTRL case flow setup times are much lower (3.5 and 9ms, respectively). So, DCP-SA provides flow setup times very close to N-CTRL (hypothetical lower bound for flow setup time) case and also incurs messaging overhead very close to 1-CTRL (lower bound for messaging overhead) case. On the other hand, DCP-GK does not provide as good result as DCP-SA, but the solutions are quite good and require fractions of seconds to run.

## VII. Conclusion

In this paper, we identified the Dynamic Controller Provisioning Problem (DCPP) in SDN. We proposed a management framework for dynamically deploying multiple controllers. We also provided a mathematical formulation of DCPP as an ILP. Since DCPP is an $\mathcal{NP}$-hard problem, we provided two heuristic algorithms (DCP-GK and DCP-SA) to solve it. The emulation results presented in this paper provide important insights on various controller placement strategies. Running a single controller causes high flow setup delay, as propagation delay between controller and switches are higher and flow setup requests can get queued at the controller because of limited processing capacity. On the other hand, running one controller per switch can provide close to zero flow setup times, but incurs significant overhead for inter-controller communication. Our framework achieves a balance between flow setup time and messaging overhead. Emulation results show that, DCP-SA and DCP-GK succeed to find a right trade-off between these two extremes and provide near optimal solutions. DCP-SA provides better results than DCP-GK, but takes longer to converge.

We intend to extend this work in three directions. First, we want to improve the convergence time of DCP-SA. An interesting approach is to generate quick but less accurate initial solutions using DCP-GK and then optimizing them using DCP-SA. Second, we want to explore other heuristic algorithms to achieve better performance and accuracy. Third, we intend to further demonstrate the effectiveness of the proposed management framework through experiments on a real testbed.

## References


[1] https://github.com/noxrepo/pox.

[2] https://github.com/noxrepo/nox.

[3] http://iperf.sourceforge.net.

[4] Ansi t1.tr.68-2001 enhanced network survivability performance.

[5] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: scaling flow management for high-performance networks. In *SIGCOMM 2011*, pages 254–265.

[6] S. Gebert, R. Pries, D. Schlosser, and K. Heck. Internet access traffic measurement and analysis. In *Traffic Monitoring and Analysis*, volume 7189 of *LNCS*, pages 29–42. 2012.

[7] S. Hassas Yeganeh and Y. Ganjali. Kandoo: a framework for efficient and scalable offloading of control applications. In *HotSDN 2012*, pages 19–24.

[8] B. Heller, R. Sherwood, and N. McKeown. The controller placement problem. In *Proceedings of HotSDN 2012*, pages 7–12, 2012.

[9] S. G. Kolliopoulos and C. Stein. Improved approximation algorithms for unsplittable flow problems. In *FOCS 1997*, pages 426–436.

[10] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: a distributed control platform for large-scale production networks. In *OSDI 2010*, pages 1–6.

[11] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *HotNets 2010*, pages 19:1–19:6.

[12] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann. Logically centralized?: state distribution trade-offs in software defined networks. In *HotSDN*, pages 1–6, 2012.

[13] N. Spring, R. Mahajan, and D. Wetherall. Measuring isp topologies with rocketfuel. In *SIGCOMM 2002*, pages 133–145.

[14] A. Tootoonchian and Y. Ganjali. HyperFlow: a distributed control plane for OpenFlow. In *NM/WREN 2010*, pages 3–3.

[15] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood. On controller performance in software-defined networks. In *Hot-ICE*, pages 10–10, 2012.

[16] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with DIFANE. In *SIGCOMM 2010*, pages 351–362.