BalCon: A Distributed Elastic SDN Control via Efficient Switch Migration

Marco Cello*, Yang Xu†, Anwar Walid‡, Gordon Wilfong‡, H. Jonathan Chao† and Mario Marchese§
* *Nokia Bell Labs, Dublin, Ireland. Email: marco.cello@nokia-bell-labs.com*
†*NYU Tandon School of Engineering, New York, NY, USA. Email: yang@nyu.edu, chao@nyu.edu*
‡*Nokia Bell Labs, Murray Hill, NJ, USA. Email: anwar.walid@nokia-bell-labs.com, gordon.wilfong@nokia-bell-labs.com*
§*University of Genoa, Genoa, Italy. Email: mario.marchese@unige.it*

***Abstract*—Scalability and reliability are among the main concerns in large-scale Software Defined Networking (SDN) application scenarios. A common approach is to use multiple distributed controllers, each managing one *static* partition of the network. In this paper, we show that dynamic mapping can improve efficiency in managing traffic load variations. We then propose BalCon (Balanced Controller): an algorithmic solution designed to tackle and reduce the load imbalance among SDN controllers through proper SDN switch migrations. Simulations demonstrate that BalCon is lightweight from the computational point of view and reduces the load imbalance among SDN controllers (expressed as variance) by 40% by migrating only a small number of switches. We also built a realistic prototype of SDN controller, BalConController, based on the open-source SDN framework RYU.**

***Keywords*-software-defined networking, distributed controllers, load balancing, multi-way partitioning**

I. Introduction

As with other centralized systems, the use of a single controller in large-scale Software Defined Networks (SDN) brings up issues of poor scalability and reliability. As the number of SDN switches managed by the controller increases, the SDN controller may fail to process all the requests coming from the switches. Moreover, because of the single point of failure, in case of malfunction of the SDN controller, all switches will become unavailable. Recent works have proposed the use of multiple physically distributed SDN controllers to improve performance scalability and reliability, while preserving the simplicity of the centralized system [18], [19], [25].

The problem of current multicontroller architectures is that they rely on a statically configured mapping between SDN switches and controllers that makes the control plane unable to adapt to traffic variation. As suggested in [9], real networks may exhibit huge variations in both temporal dimensions (traffic varies at different time of the day or even in a shorter time scale) and spatial dimensions (traffic varies at different locations of the network) [6]. If the SDN switch-controller mapping is static, a controller may become overloaded while others are underutilized.

An overloaded controller will response to the switches with an increased latency, deteriorating the Quality of Service. In this scenario, as load imbalance occurs, it would be desirable to have a dynamic mapping between the controllers and the switches. An overloaded SDN controller should have the ability to migrate a subset of its switches to other controllers in order to reduce its congestion.

This paper provides theoretical, algorithmic and implementation contributions which are summarized below:

- we show that dynamic mapping between SDN switches and controllers provides system elasticity and efficiency during traffic load variations;
- we model the problem of switch migrations among SDN controllers to achieve controller load balancing as an optimization problem and we demonstrate that the problem is NP-complete;
- since the optimal solution, for the aforementioned switch migration problem, is impractical due to the prohibitively high computational complexity, we propose BalCon, a heuristic solution that is able to maintain load balancing among SDN controllers, through SDN switch migration, even under dynamic traffic load;
- we implement BalCon in Matlab, and the simulation results show that BalCon reduces load imbalance among controllers (expressed as variance of the load) by 40% and reduces the load of the congested controller by 19% with a relatively low number of SDN switches migrated; we also compare BalCon with METIS a graph partition heuristic that tries to minimize the sum of weight cut among partitions. BalCon slightly underperforms when compared to METIS, but requires fewer switch migrations.
- we build a realistic prototype of SDN controller - BalConController - based on RYU (a popular SDN controller written in python [4]) that is extended with new functions supported: multicontroller, switch migration procedure and BalCon algorithm.

The rest of the paper is organized as follows: Section II presents the motivations of our work. Section III presents the system model. Section IV presents the design and the details of BalCon. In Section V we evaluate BalCon. Section VI presents a realistic prototype of a SDN controller. Section VII reviews prior related works. Conclusions are in Section VIII.

978-1-5090-5817-4/17 $31.00 © 2017 IEEE
DOI 10.1109/IC2E.2017.33

II. Motivations

An SDN network is composed of SDN switches and a logically centralized SDN controller. Each SDN switch processes and delivers packets according to rules stored in its flow table (forwarding state), whereas the SDN controller configures the forwarding state of each switch using a standard protocol (e.g., OpenFlow [21]). The SDN controller is also responsible for constructing the virtual topology (i.e., a graph) representing the physical topology. Virtual topology is used by application modules that run on top of the SDN controller to implement different control logics and network functions (e.g., routing, traffic engineering, firewall). Traffic rules, representing the forwarding state, are installed in SDN switches when a new flow arrives[1].

In order to overcome the scalability issues of a single centralized controller, several approaches have been proposed in the literature. One of the most effective methods is the use of distributed controllers. Existing distributed controller solutions still suffer from the static mapping between SDN switches and controllers, limiting the capability of dynamic load adaptation.

Let's briefly explain the reactive mode behaviour in SDN. In Figure 1, a new flow $f1$ generated by host H_1 arrives at S_1. S_1 doesn't have any rule associated with the flow and generates a "packet-in"[2] to C_1 (i.e., the first green arrow). C_1 then computes the route (i.e., the line in red) and installs the flow rules on SDN switches controlled by itself (i.e., the blue arrows to S_1 and S_2). When the flow arrives at S_5, the switch doesn't have any rule associated with the flow and, consequently, sends a packet-in request to C_2 that computes the flow's path and installs the flow rules on S_5 and S_6.

Suppose now that due to the traffic variations, a large number of new flows arrive to the network and the current traffic pattern is depicted in Figure 1. In particular:

- host H_1 generates 30 new $flows/second$ to H_3, which are routed through $S_1 \rightarrow S_3 \rightarrow H_3$ (green arrows);
- host H_2 generates 35 new $flows/second$ to H_6, which are routed through $S_2 \rightarrow S_5 \rightarrow S_6 \rightarrow H_6$ (red arrows);
- host H_8 generates 20 new $flows/second$ to H_2, which are routed through $S_8 \rightarrow S_7 \rightarrow S_4 \rightarrow S_2 \rightarrow H_2$ (blue arrows).

At this point, we want to ask what are the computational burdens of SDN controllers C_1 and C_2 due to the instantiation of the new flows. Suppose that the path computation for a single flow requires α unit of load at the controller, whereas the rules installation of a single flow in a single switch requires β units of load at the controller. At controller C_1:

- the green flows generate 30α units for path computation and $(30+30)\beta$ units for rules installation at S_1 and S_3;
- the red flows generate 35α units for path computation and 35β units for rules installation at S_2;
- the blue flows generate 20α units for path computation and $(20+20)\beta$ units for rules installation at S_4 and S_2.

At controller C_2:

- the red flows generate 35α units for path computation and $(35+35)\beta$ units for rules installation at S_5 and S_6;
- the blue flows generate 20α units for path computation and $(20+20)\beta$ units for rules installation at S_8 and S_7.

If we assume that the path computation load is larger than the rule installation load, e.g., $\alpha = 1$, $\beta = 0.1$[3], we obtain:

$$L_{C_1} = (30+35+20)\alpha + (30+55+30+20)\beta = \\ = 98.5\ units/s.$$
$$L_{C_2} = (35+20)\alpha + (35+35+20+20)\beta = \\ = 66\ units/s.$$

In the aforementioned example, the load between controllers C_1 and C_2 is highly unbalanced if the mapping between the controllers and switches is static. If we have the capability to dynamically shrink or enlarge the SDN domains or partitions through a proper switch migration, we can obtain the new mapping between controllers and switches in Figure 2.

We observe in Figures 2 that S_2 and S_4 are now part of the second domain and controlled by C_2. Since C_1 does not manage any longer S_2 and S_4, the computation burden due to the red and blue flows is only managed by C_2. The new

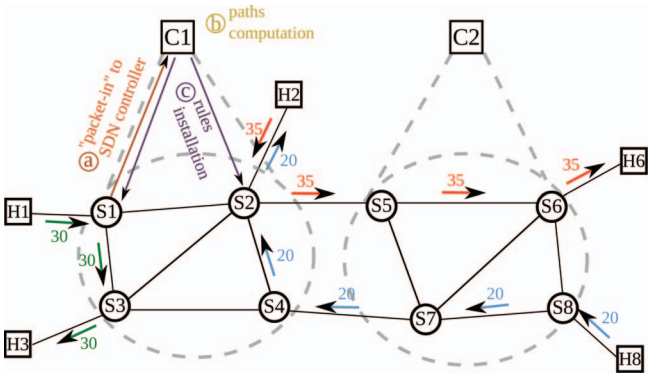

Figure 1. SDN controller load imbalance scenario.

[1]This method is known as "reactive" mode. A less-used and less-effective method is "proactive" mode in which the controller installs rules beforehand.

[2]When a packet does not match any of the existing rules inside an SDN switch, the default policy is to send a copy of that packet up to the controller. This "packet sent to the controller" message is called, in OpenFlow-parlance, a packet-in [22].

[3]Here we consider the path computation load, ten times larger than the rules installation load. The estimation of those values are not the main focus of this paper.

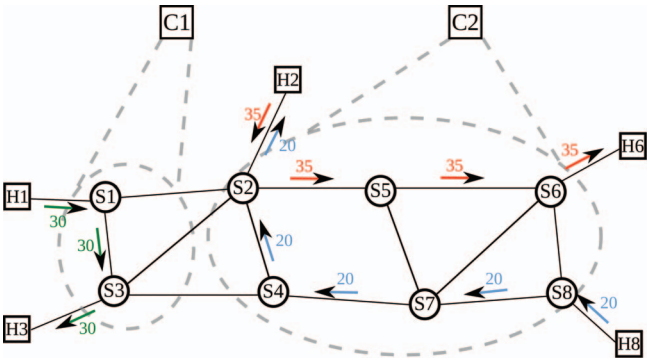

Figure 2. Controller load balance is improved after switch migrations.

controllers' load are now:

$$L_{C_1} = (30)\alpha + (30+30)\beta = \\ = 36\ units/s.$$
$$L_{C_2} = (35+20)\alpha + (55+20+35+35+20+20)\beta = \\ = 73.5\ units/s.$$

Therefore, we obtained a significant reduction of the controller load at C_1 (63%) compared to a relatively small increase of the controller load at C_2 (11%).

As explained in [9], using real measurements of a production datacenter the authors in [6] found that there are 1-2 orders of magnitude difference between peak and median flow arrival rates at the switch: peak flow arrival rate can be up to 300M/s with the median rate between 1.5M/s and 10M/s. Assuming that each controller can manage up to 2M/s as flow arrival rate, it requires only 1-5 controllers to process the median load, but 150 for peak load. If we use static mapping, each controller needs to have the capacity to process the peak flow arrival (worst-case situation). If we have a dynamic mapping, the capacity of each controller can be lowered, since the peak of different partition (domain) usually will not occur at the same time due to multiplexing and sharing effect.

Recent work, [20], shows that current single controllers can manage up to 20M/s as flow arrival rate.

Motivated by the above observations, we seek to answer our key question: how to dynamically select and migrate switches from the domain of one controller to another to balance controller load? The quality of this answer will largely depend on the complexity and the cost of the switch migration process.

We first develop optimal controller load balancing (CLB) problem in SDN multicontroller scenarios, and prove, however, that it is NP-Complete problem. We then model the CLB problem as a graph partitioning problem and develop BalCon: an effective algorithm for load adaptation among SDN controllers through SDN switch migrations.

III. Modeling of Controller Load Balancing Problem

A. System Model

The objective of this section is to find an appropriate model that takes into account the flow arrival dynamics at each SDN switch and relate them to the computational load at each SDN controller. We then formalize the CLB problem into an optimization one.

An SDN scenario is composed of a set $\mathcal{S}$ of SDN switches, $S_i \in \mathcal{S}$, managed by a set $\mathcal{C}$ of SDN controllers, $C_m \in \mathcal{C}$. In accordance with prior works, we cannot assume predictable traffic or well-known traffic patterns among the SDN switches, but we can monitor the traffic load during runtime. Therefore, we indicate with f_{o,S_i} the current arrival rate of new flows at SDN switch S_i from outside the SDN network, with $f_{S_i,o}$ the current arrival rate of new flows that leave the SDN network from switch S_i, whereas with f_{S_i,S_j} we indicate the current arrival rate of new flows traversing the link between the two connected SDN switches S_i and S_j. In other words, f_{S_i,S_j} represents the current arrival rate of new flows at the SDN switch S_j coming from SDN switch S_i. Referring to Figure 1 we have: $f_{o,S_1} = 30$, $f_{o,S_2} = 35$, $f_{o,S_8} = 20$, $f_{S_3,o} = 30$, $f_{S_2,o} = 20$, $f_{S_6,o} = 35$, $f_{S_1,S_3} = 30$, $f_{S_2,S_5} = 35$, $f_{S_4,S_2} = 20$, $f_{S_5,S_6} = 35$, $f_{S_7,S_4} = 20$, $f_{S_8,S_7} = 20$.

As shown before, the load L_{C_m} at controller C_m is composed of three main components: the path computation load of new flows arriving from outside the SDN network (e.g., green arrow $H_1 \rightarrow S_1$ and red arrow $H_2 \rightarrow S_2$ in Figure 1); the path computation load of the flows arriving from another SDN domains (e.g., blue arrow $S_7 \rightarrow S_4$ in Figure 1); the rule installation load at each switch controlled by C_m for all flows traversing the domain controlled by C_m.

Definition 3.1: - Path Computation Load for External Flows - When a batch of flows arrive at S_i from outside the network with a rate of f_{o,S_i}, they generate a computational load due to the *path computation* at the SDN controller of S_i equal to:

$$\mathcal{K}(f_{o,S_i}) \tag{1}$$

Definition 3.2: - Path Computation Load of flows from Other SDN Domains - When a batch of flows arrive at S_i from S_j, a switch controlled by another SDN controller, with a rate of f_{S_j,S_i}, they generate a computational load due to the *path computation* at the SDN controller of S_i equal to:

$$\mathcal{K}(f_{S_j,S_i}) \tag{2}$$

The computational load at SDN controller necessary to perform path computation is dependent on the arrival rate of flows through a function $\mathcal{K}$. The definition of the function $\mathcal{K}$ is not the objective of this work.

Definition 3.3: - Rules Installation Load - The computational load at the controller due to rules installation in switch

S_i is equal to:

$$\sum_{S_j \in \mathcal{S}} \mathcal{G}(f_{S_i,S_j}) + \mathcal{G}(f_{S_i,o}) \tag{3}$$

Equation 3 expresses the amount of flows that are traversing S_i going to other switches or out of the SDN network. Function $\mathcal{G}$ maps the the flow arrival rate at S_i to the computational load at the SDN controller needed for rules installation.

Definition 3.4: The set of SDN switches controlled by SDN controller C_m is denoted by $\mathcal{P}_m$.

The set $\mathcal{S}$ is then partitioned in a $|\mathcal{C}|$-partition, with $\mathcal{P}_m \subset \mathcal{S}$, $\mathcal{P}_m \cap \mathcal{P}_n = \emptyset,\ n \neq m$.

Definition 3.5: The overall computational load at SDN Controller C_m (L_{C_m}) is computed as:

$$\begin{aligned} L_{C_m} \triangleq & \sum_{S_i \in \mathcal{P}_m} \mathcal{K}(f_{o,S_i}) + \sum_{\substack{S_j \notin \mathcal{P}_m \\ S_i \in \mathcal{P}_m}} \mathcal{K}(f_{S_j,S_i}) + \\ & + \sum_{\substack{S_i \in \mathcal{P}_m \\ S_j \in \mathcal{S}}} \mathcal{G}(f_{S_i,S_j}) + \sum_{S_i \in \mathcal{P}_m} \mathcal{G}(f_{S_i,o}) \end{aligned} \tag{4}$$

Overloading the SDN controller reduces its responsiveness and causes a performance degradation since the flows will experience an unexpected latency.

Definition 3.6: An SDN controller is overloaded or congested when its overall computational load is:

$$L_{C_m} > \boldsymbol{L} \tag{5}$$

where $\boldsymbol{L}$ that indicates the maximum computational load tolerated at each SDN controller.

When congestion occurs a migration procedure is needed to reduce overload. In particular, starting from a partition $(\mathcal{P}_1, \ldots, \mathcal{P}_{|\mathcal{C}|})$ for which at least one controller, C_m, the condition $L_{C_m} > \boldsymbol{L}$ holds, we need to find a new partition $(\mathcal{P}'_1, \ldots \mathcal{P}'_{|\mathcal{C}|})$ such that the SDN controller load $L_{C_m} \leq \boldsymbol{L}$, $C_m \in \mathcal{C}$.

The controller load balancing (CLB) problem can be expressed as a mathematical optimization problem which we call the Optimal CLB (OCLB) problem, and is defined as follows:

Definition 3.7: - OCLB Problem.

$$\min_{\mathcal{P}_1,\ldots,\mathcal{P}_{|\mathcal{C}|}} \max_{C_m \in \mathcal{C}} L_{C_m}; \tag{6}$$

subject to

$$\mathcal{P}_m \cap \mathcal{P}_m = \emptyset,\ m \neq n,;$$
$$\bigcup \mathcal{P}_m = \mathcal{S}.$$

B. CLB as Graph Partitioning Problem

The CLB problem can be expressed as a partitioning problem on a graph and the computation of L_{C_m} can be induced directly on the graph.

In particular, we represent the SDN network as a directed edge-weighted and vertex-weighted graph $G(\mathcal{S}, \mathcal{E})$ in which SDN switches are the vertices with weights $l(S_i), S_i \in \mathcal{S}$ and edges $\mathcal{E} = \{(S_i, S_j) : S_i, S_j \in \mathcal{S}, l(S_i, S_j) > 0\}$, are the connections among SDN switches. $l(S_i, S_j)$ is the edge weights of (S_i, S_j). That is

$$l(S_i) = \mathcal{K}(f_{o,S_i}) + \sum_{S_j \in \mathcal{S}} \mathcal{G}(f_{S_i,S_j}) + \mathcal{G}(f_{S_i,o}); \tag{7}$$

$$l(S_j, S_i) = \mathcal{K}(f_{S_i,S_i})\ . \tag{8}$$

The overall load at C_m, denoted by L_{C_m}, is then the sum of the weights of the vertices belonging to its partition plus the sum of weights of the edges directed to the partition of C_m. Specifically:

$$L_{C_m} = \sum_{S_i \in \mathcal{P}_m} l(S_i) + \sum_{\substack{S_j \notin \mathcal{P}_m \\ S_i \in \mathcal{P}_m}} l(S_j, S_i). \tag{9}$$

Note that Equation 9 is just another expression for Equation 4.

Figure 3 is a representation of Figure 1 as a graph partitioning problem. For example, the vertex weight of S_1 represent the computational load "brought" by S_1 to C_1. In particular $l(S_1) = 33$, which is the sum of $\mathcal{K}(f_{o,S_1}) = 30$ ($30\ flows/s$) and the rule installation for the flows going to S_3 $\mathcal{G}(f_{S_1,S_3}) = 3$[4].

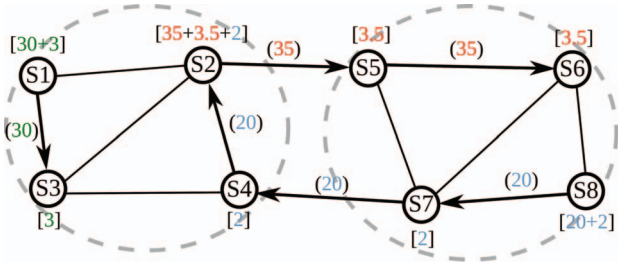

Figure 3. The SDN network scenario of Figure 1 as graph partitioning problem.

Refering to the same figure we get:

$$\begin{aligned} L_{C_1} &= l(S_1) + l(S_2) + l(S_3) + l(S_4) + l(S_7, S_4) = \\ &= 33 + 40.5 + 3 + 2 + 20 = 98.5\ units/s. \\ L_{C_2} &= l(S_5) + l(S_6) + l(S_7) + l(S_8) + l(S_2, S_5) = \\ &= 3.5 + 3.5 + 2 + 22 + 35 = 66\ units/s. \end{aligned}$$

C. Sketch of NP-completeness Proof

In what follows we have a network $H = (W, A)$ where the nodes in W represent routers and arcs in A are symmetric (i.e., $(u, v) \in A$ if and only if $(v, u) \in A$. A flow f in a network H is a directed acyclic path in H. Consider a partition of W into regions $R_1, R_2, \ldots, R_c$. For a flow

[4]For simplicity here we consider the functions $\mathcal{K}$ and $\mathcal{G}$ as linear functions of the rate: $\mathcal{K}(rate) = rate$, $\mathcal{G}(rate) = rate/10$.

f we cut f into maximal sections contained in a region $f_1, f_2, \ldots, f_t$. That is, f_i and f_{i+1} are contained regions R_a and R_b respectively where $R_a \neq R_b$. We then say that f *starts* at the first router (node) in each f_i.

Given a set of flows F, a bound B on the number of flows that can start at a router and a bound K on the total number of flows starting at routers within any given region, we say that a partition of W is *valid* if no router or controller bound is exceeded. We call the problem of deciding if a valid partioning exists, the VALID PARTITIONING problem.

Theorem 3.8: The VALID PARTITIONING problem is NP-complete.

Proof: Consider the MINIMUM CUT INTO EQUAL-SIZED SUBSETS problem where one is given an unweighted graph $G = (V, E)$, two vertices s and t, a bound d and the question is whether there exists a partitioning of V into two equal parts V_s and V_t where $s \in V_s$, $t \in V_t$ so that at most d edges cross between V_t and V_s. The MINIMUM CUT INTO EQUAL-SIZED SUBSETS problem is known to be NP-complete [12]. We show that VALID PARTITIONING is NP-hard using a reduction from MINIMUM CUT INTO EQUAL-SIZED SUBSETS.

Details of reduction are omitted due to space limitations. The complete proof can be found here: **NP-completeness Proof**[5]. ■

IV. BALCON ALGORITHM

An optimal SDN switch migration is impractical due to its computational complexity (i.e., OCLB problem is NP-complete) and could lead to undesirable excessive switch migrations. A more practical approach should involve incremental adjustment of the switch partitions, i.e., only a small number of SDN switches are migrated.

In this section, we propose *Balanced Controllers* (BalCon), an algorithmic solution designed to tackle and reduce the load imbalance among SDN controllers through a proper SDN switch migration. The key observation behind *BalCon* is that an effective switch migration can be based on analysis of the communication patterns of the SDN switches. The switch migration should be at the granularity of *clusters*: switches with strong connections[6] should always be assigned to the same controller.

> Analysis of the traffic pattern among SDN switches helps identify clusters of connected switches. Migration at the granularity of the cluster (i.e., cluster migration) can reduce the work load at the controllers, as compared to single switch migration.

BalCon is an heuristic algorithm which operates during the network runtime and is able to detect and solve congestion at the SDN controllers through proper SDN switch migrations. BalCon can be implemented as a northbound application of the SDN controller (more details are available in Section VI). BalCon consists of three phases, as summarized below:

1. Monitoring and congestion detection: During the network operation, BalCon continuously monitors the congestion level at each SDN controller. An SDN controller, C_m, is considered congested when L_{C_m} reaches a predetermined threshold. BalCon then computes a list of SDN switches that may be migrated. The list is ordered by a priority computed using a pre-determined metric. For example, the SDN switches that are observing a rapid increase of new flows could get high priority since they could rapidly overload the SDN controller with packet-ins.

2. Clustering and migration evaluation: Starting from the SDN switches in the priority list, BalCon analyzes the traffic pattern among SDN switches to find clusters of heavily connected switches (discussed below).

3. Cluster migration: When the best cluster is found and the migration is evaluated, the SDN switches belonging to the cluster are migrated to the new SDN controller.

The algorithm we propose is substantially based on the repetition of three functions: $IncreaseCluster$ in which the cluster is expanded; $ComputeMigrationAlternatives$ in which the migrations on different SDN controllers of the current cluster are evaluated (producing the "migration alternatives" or simply called "alternatives" in the following); $Evaluate\text{-}BestMigrationAlternative$ in which given a list of alternatives, the best alternative (based on some criteria described in the following) is computed. The algorithm is shown in Algorithm 1.

Algorithm 1: BalCon

Input: Edge- and node-weighted graphs $G(\mathcal{S}, \mathcal{E})$, congested SDN controller C_m;

```
P_m: set of SDN switches controlled by the congested SDN controller C_m;
A = ComputeStartingSwitchesList(C_m)
foreach S_i ∈ A do
    T = {S_i};
    alternatives = alternatives ∪ ComputeMigrationAlternatives(T);
    while 1 do
        newT = IncreaseCluster(T);
        if size(T) > mcs || newT = T then
            break;
        T = newT;
        alternatives = alternatives ∪ ComputeMigrationAlternatives(T);
[T^0, Target SDN controller^o] ← EvaluateMigrationAlternatives(alternatives);
```

From the set $\mathcal{P}_m$ (SDN switches controlled by the congested SDN controller C_m), the algorithm extracts a subset

[5]https://marcocello.github.io/pubs/IC2E2017-BalCon-Proof.pdf

[6]We consider the relative density of the cluster [24].

list $\mathcal{A}$ (*StartingSwitch List*) that contains the starting nodes used for the cluster construction (line 2). $\mathcal{A}$ could be computed, for example, by looking for the SDN switches that have a significant increase in flow arrival rate. The first SDN switch belonging to $\mathcal{A}$ is selected and inserted in the empty cluster $\mathcal{T}$ (Line 4). The migration alternatives of the SDN switches belonging to $\mathcal{T}$ are computed through $ComputeMigrationAlternatives$. The algorithm, subsequently, executes a while loop in which the cluster is continuously enlarged with the $IncreaseCluster$ function and evaluated with the function $ComputeMigration$-$Alternatives$. The algorithm halts when one of the two stop conditions are met: the cluster reaches a predetermined size mcs (max cluster size), i.e., $size(\mathcal{T}) > mcs$, or the increased cluster is equal to the old one ($new\mathcal{T} = \mathcal{T}$). The next switch in $\mathcal{A}$ is then selected and inserted in an empty cluster $\mathcal{T}$. When the $mssls$ (max starting switch list size) is reached, all the migration alternatives are evaluated using the $AlternativeEvaluation$ function. The best alternative composed by $\mathcal{T}^0$ (the cluster) and the target SDN controller (the controller that will receive $\mathcal{T}^0$) are chosen and the migration can occur. In the following we will give a detailed explanation of the aforementioned functions.

```
1 function ComputeMigrationAlternatives (T);
2 foreach SDN controller C_i do
3     "virtual" migrate cluster T in SDN controller C_i;
4     if L_{C_i} < L then
5         compute L_{C_n}, ∀C_n ∈ C;
6         compute migrationSize for this new configuration;
7         save them in lastAlternatives
8 return lastAlternatives
```

$ComputeMigrationAlternatives$ "virtual" migrates cluster $\mathcal{T}$ in different SDN controller destinations. For each controller, it computes the controller load and the migration size. Table I shows a possible output of $ComputeMigration$-$Alternatives$ routine in a scenario with 60 switches and 5 controllers, when $\mathcal{T} = \{S_1,S_2,S_{56}\}$. For SDN controller C_i, the function migrates $\mathcal{T}$ to SDN controller C_i (Line 3), computing the new computational load at each SDN controller (Line 5) and the migration cost $migrationSize$ (Line 6) defined as the number of switches that need to be migrated.

```
1 function IncreaseCluster(T);
2 neighborsT = ComputeNeighborsOfCluster(T);
3 foreach S_i ∈ neighborsT do
4     newT = T ∪ S_i;
5     savedDensities = [savedDensities; S_i, Density(newT)];
6 S_i^o = argmax_{savedDensities} Density(newT) ;
7 return T ∪ S_i^o;
```

Starting from the cluster $\mathcal{T}$, the function constructs the set $neighbors\mathcal{T}$ composed of all SDN switches that are neighbors to $\mathcal{T}$. An SDN switch S_i is a neighbor of $\mathcal{T}$ if $\exists S_j \in \mathcal{T} : l(S_i, S_j) \neq 0, l(S_j, S_i) \neq 0$. The function then selects the neighbor that maximizes the relative density $Density$ [24] of the newly created cluster.

$\mathcal{T}$	Target SDN Controller	$[L_{C_1}, \ldots, L_{C_{\lvert\mathcal{C}\rvert}}]$	$migration\ size$
$\{S_1,S_2,S_{56}\}$	C_1	[90, 9, 6, 10, 8]	0
$\{S_1,S_2,S_{56}\}$	C_2	[86, 51, 6, 10, 8]	3
$\{S_1,S_2,S_{56}\}$	C_3	[70, 9, 48, 10, 8]	3
$\{S_1,S_2,S_{56}\}$	C_4	[80, 9, 6, 51, 8]	3
$\{S_1,S_2,S_{56}\}$	C_5	[96, 9, 6, 10, 50]	3

Table I
EXAMPLE OF *alternatives* CARRIED OUT BY BALCON ALGORITHM IN A TOPOLOGY WITH 50 SWITCHES, 5 CONTROLLERS AND A CLUSTER $\mathcal{T} = \{S_1,S_2,S_{56}\}$.

Definition 4.1: Relative density is the ratio of the internal degree to the number of incident edges, i.e.,

$$Density(\mathcal{T}) = \frac{\sum\limits_{\substack{S_i,S_j \in \mathcal{T} \\ S_i \neq S_j}} l(S_i, S_j)}{\sum\limits_{\substack{S_i,S_j \in \mathcal{T} \\ S_i \neq S_j}} l(S_i, S_j) + \sum\limits_{\substack{S_i \in \mathcal{T} \\ S_j \in \mathcal{S} \setminus \mathcal{T}}} l(S_i, S_j)} \quad (10)$$

Given the $alternatives$ vector, $EvaluateMigrationAlternatives$ chooses the best alternative ([$\mathcal{T}^o$,Target SDN controller^o]) among them, that optimizes one of the following $Evaluation$-$Method$:

minMax - Minimize the maximum controllers load:

$$\underset{alternatives}{\operatorname{argmin}} \left(\max\ [L_{C_1}, \ldots, L_{C_{|\mathcal{C}|}}] \right) \quad (11)$$

minSum - Minimize the sum of controllers load:

$$\underset{alternatives}{\operatorname{argmin}} \sum_{C_m \in \mathcal{C}} L_{C_m} \quad (12)$$

integral - Maximize the distance from the controllers load configuration in case of congestion:

$$\underset{alternatives}{\operatorname{argmax}}\ \mathcal{D}([L_{C_1}, \ldots, L_{C_{|\mathcal{C}|}}], [\widehat{L}_{C_1}, \ldots, \widehat{L}_{C_{|\mathcal{C}|}}]) \quad (13)$$

with $[\widehat{L}_{C_1}, \ldots, \widehat{L}_{C_{|\mathcal{C}|}}]$ the vector of controllers load when congestion appears just before BalCon, and function $\mathcal{D}(\mathbf{u}, \mathbf{v})$ defined as follow:

$$\mathcal{D}(\mathbf{u}, \mathbf{v}) = \sum_i \int_{u_i}^{v_i} x^2 dx \quad (14)$$

V. Performance Evaluation

BalCon has been implemented using Matlab R2015a 64bit for Linux. The simulations has been carried out using a PC equipped with an Intel Core i5-3340@3.10 GHz with 8 GB of 1600 MHz DDR3 RAM and an OS Linux Mint 17.

A. Dynamic Scenario - Effectiveness of BalCon

In Dynamic Scenario simulations set we fix BalCon parameters (mcs, $mssls$ and $EvaluationMethod$) and we evolve the network over time in order to show the effectiveness of BalCon during a (simulated) runtime network operation. We simulated 4 different network topologies shown in Table II, varying the degree in which edge-core (dEC) and core-core (dCC) nodes are connected. In particular, dEC represents the number of connections that each edge node has towards core nodes, while each dCC represents the number of connections each core node has towards other core nodes. To perform Dynamic Scenario simulations we implemented a routine that generates flow arrivals and departures at edge nodes following a Poisson process. For each topology presented, we run 200 different simulations with different seeds of the Poisson process generator. Each run simulates $2000s$ of network runtime operation. BalCon has been setup using a starting switch list size $mssls = 20$ and a maximum cluster size $mcs = 20$ using Equation 13 (Integral) as $EvaluationMethod$ in $EvaluateMigrationAlternatives$.

Name	# edge	# core	dEC	dCC	# controllers
$Topology1$	50	40	2	full mesh	5
$Topology2$	50	40	5	full mesh	5
$Topology3$	50	40	2	$\frac{\#\text{ core nodes}}{5}$	5
$Topology4$	50	40	5	$\frac{\#\text{ core nodes}}{5}$	5

Table II
Topologies simulated for performance analysis.

Figure 4 shows a topology composed of 9 edge nodes (in blue), 5 core nodes (in gray), and 3 controllers. $dEC = 1$ indicates that each edge node is connected to a single core node, while $dCC =$ full mesh since the core nodes form a full mesh network.

Figure 5 shows the computational load of all the 5 controllers (0 means no congestion at all, while 100 indicates overload) during the simulation of $Topology1$.The green line represents the congestion level of controller C_5. As soon as it reaches the threshold $\boldsymbol{L} = 90$, BalCon is triggered using the starting switch list size $swlsm = 20$ and the maximum cluster size $msc = 20$. The different routines of BalCon are indicated with black dotted ellipse.

BalCon performs well: the maximum computational load during the 4 BalCon instances is reduced on average by

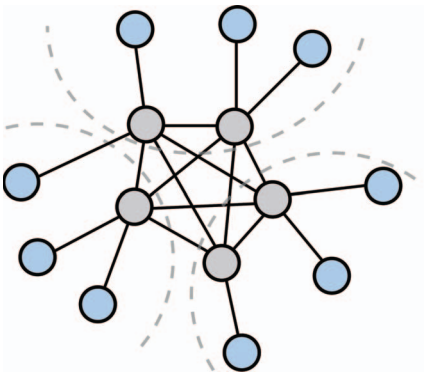

Figure 4. Example of network topology with 9 edge nodes (in blue), 5 core nodes (in gray), 3 controllers, $dEC = 1$ and $dCC =$ full mesh.

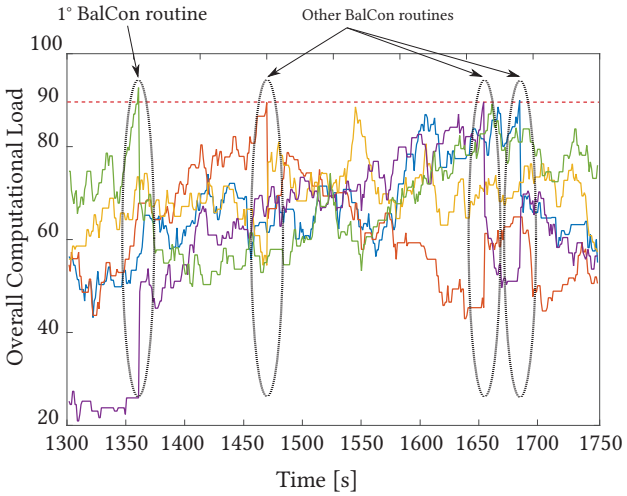

Figure 5. Computational load of 5 controllers during Dynamic Scenario and the effect of BalCon algorithm in simulations with $Topology1$ and $seed = 1$. The blue line is L_{C_1}, the red line is L_{C_2}, the yellow line is L_{C_3}, the violet line is L_{C_4} and the green line is L_{C_5}.

15%, with an average of 2.4 switches migrated in each routine. The computational time is $0.69s$. The variance of the computational load is reduced at each routine on average by 66%. In this case BalCon can effectively balance the computational load and solve the overloading problem at the controller with few switch migrations.

Figure 6 clearly shows the performance advantage of BalCon algorithm compared to the static assignment of the switches to the controller using the same traffic pattern. Figure 6(a) shows the computational load of the 5 controllers without load balancing, i.e., static assignment, while Figure 6(b) is the case in which BalCon is implemented. As we observe, BalCon maintains the controllers' load below the threshold during runtime, whereas in the static assignment case the congestion load exceeds the threshold (90) by 50%. Other settings with different topologies in Table II show similar results as Figure 6.

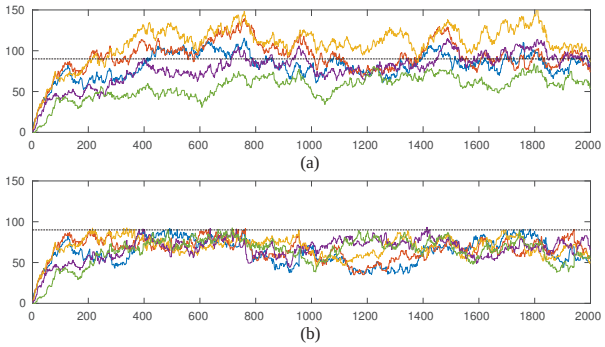

Figure 6. Comparison of the computational load between a static assignement (a) and BalCon (b) in Dynamic scenario with $Topology3$.

B. Static Scenario

In Static Scenario simulations set we fix the time instant (when congestion occurs) and we vary BalCon parameters in order to show how the parameters affect BalCon's performance. We varied the starting switch list size ($mssls$), the maximum cluster size (mcs) and the method for $EvaluateMigrationAlternatives$ function. We simulated 4 different network topologies shown in Table II. For each topology we synthetically generated 500 different "congestion traffic configurations" in which one controllers is congested. For each congestion traffic configuration we run several instances of BalCon algorithm varying $mssls = \{3, 5, 10, 20\}$ and $mcs = \{3, 5, 10, 20\}$.

For each simulation, we evaluated different performance indicators. Let $\boldsymbol{L_C} = [L_{C_1}, \dots]$ the vector denote the controllers' load, $\boldsymbol{L_C^{con}}$ the controllers' load when congestion appears just before the application of BalCon and $\boldsymbol{L_C^{bal}}$ the loads after BalCon routine.

Definition 5.1: Let the congested controller $C_m^* = \text{argmax } \boldsymbol{L_C^{con}}$ and the congested controller load $\boldsymbol{L_C^{con}}(C_m^*)$. We define the Reduction Congested Controller Load $(\%)$ as:

$$\frac{\boldsymbol{L_C^{bal}}(C_m^*) - \boldsymbol{L_C^{con}}(C_m^*)}{\boldsymbol{L_C^{con}}(C_m^*)} \cdot 100\,. \tag{15}$$

Definition 5.2: Reduction Max Controller Load $(\%)$

$$\frac{\max \boldsymbol{L_C^{bal}} - \max \boldsymbol{L_C^{con}}}{\max \boldsymbol{L_C^{con}}} \cdot 100 \tag{16}$$

Definition 5.3: Reduction Sum Controller Load $(\%)$

$$\frac{\sum \boldsymbol{L_C^{bal}} - \sum \boldsymbol{L_C^{con}}}{\sum \boldsymbol{L_C^{con}}} \cdot 100 \tag{17}$$

Definition 5.4: Reduction Variance Load $(\%)$

$$\frac{Var(\boldsymbol{L_C^{bal}}) - Var(\boldsymbol{L_C^{con}})}{Var(\boldsymbol{L_C^{con}})} \cdot 100 \tag{18}$$

Figure 7 shows the performance of different versions of BalCon by varying $mssls$ and mcs using $Topology1$ and $minMax$ as $EvaluationMethod$. In the first instance, we consider the black bars, representing the choice of parameters $[mssls, mcs] = [3, 3]$. We observe a reduction of the congested controller load by 12.55% (Figure 7(a)), a reduction of the max controllers load by 11.32% (Figure 7(b)), an almost negligible reduction of the sum of the controllers load (Figure 7(c)), a 47.10% of the reduction of the variance (Figure 7(d)). We also observe that we obtain an average migration size of 1.37 switches (Figure 7(e)) and an average BalCon computation time of $0.13s$ (Figure 7(f)). Considering now the other bars, we note that the performance is highly dependent on the parameters. If we have a larger $mssls$ and mcs, we can increase the search space of the possible solutions of BalCon. This translates to better performance. In fact, if we consider the case $[mssls, mcs] = [20, 20]$, we observe a significant increase of the performance indicators described before.
With large values of $mssls$ and mcs, we can observe a small increase of the migration size (from 1.37 to 2.05). BalCon is quite fast, in fact the computation time is lower than $1s$ ($0.84s$) with higher values of $mssls$, and mcs. As we observe, BalCon is highly efficient with low computation time and few switch migrations needed.

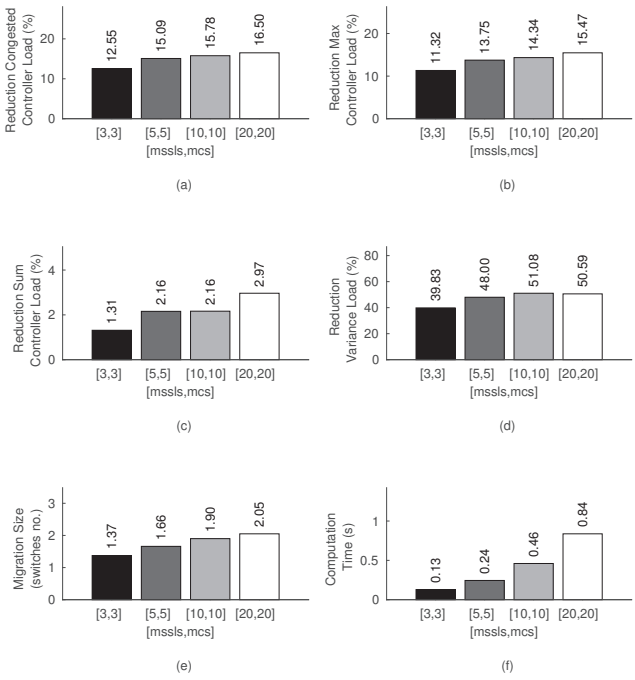

Figure 7. Performance of different version of BalCon varying $mssls$ and mcs using $Topology1$.

C. Static Scenario - N-Iterations vs METIS

In the last set of simulations, our aim is to compare the performance of BalCon with the solution carried out by METIS [17]. METIS is a graph partition heuristic that tries to minimize the sum of weight cut among partitions. Given the good results of METIS in terms of reduction of the sum of controller load, we use it as performance benchmark. Using $Topology1$, we synthetically generated 500 different congestion traffic configurations in which we brought one of the controllers to the congestion.
For each congestion traffic configuration we run several instances of BalCon with parameters $[mssls, mcs] = [20, 20]$ and $minSum$ varying the number of iterations of itself. In more detail for each traffic configuration we evaluated the performance of BalCon with 1 iteration (1-it), 2 iterations (2-it), 5 iterations (5-it) and "loop". With 1-it, for example, BalCon will be executed only 1 time to solve the congestion event. 1-it is the configuration used so far. With 5-it, BalCon will be executed 5 times consecutively to solve the congestion event. With "loop", BalCon will be continually executed until there is no more reduction of the most congested controller's load between two consecutive iterations. We compare different versions of Balcon with METIS-Multilevel k-way partitioning-ufactor=100.

Figure 8 shows the performance of different iterations of BalCon compared to METIS with $Topology1$.

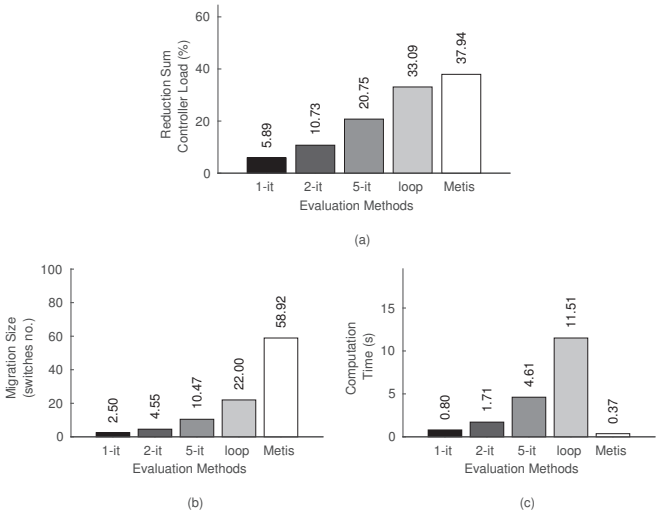

Figure 8. Performance of different iterations of BalCon compared to METIS with $Topology1$.

We can observe that BalCon loop can obtain a very good reduction of the sum of controllers load of 33.09% compared to 37.94% of METIS. METIS obtains a migration size of 58.92 switches since is completely unaware of the previous switch-controller configuration. On the other hand, BalCon loop obtains a migration size of 22 switches. The computational time is higher in BalCon (11.51 s).

[10] shows that the migration time of a single switch can take 100ms when the controller is receiving 10000 packet-in/s. Having a method that minimize the migration size, as our BalCon permits to avoid an high messaging exchange among SDN controllers that could eventually reduce the performances.

VI. Design and Real Implementation: BalConController

In this section, we present further details on how we designed and implemented BalConController by modifying and adding components to RYU controller [4].

A. Design

BalConController architecture can be implemented through a NorthBound application of the SDN controller and run in a distributed fashion: only the congested controller will activate the BalCon routine based on an updated map of the network. In particular Figure 9 shows the modules involved in the BalConController and their relationship with existing modules in a SDN controller.

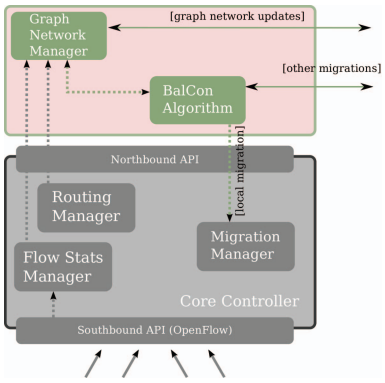

Figure 9. BalConController Architecture.

Graph Network Manager is the entity that gathers both flow arrival statistics from *Flow Stats Manager* entity and routing decisions from *Routing Manager* entity in order to construct and update the local version of the graph representation $G(\mathcal{S}, \mathcal{E})$. $G(\mathcal{S}, \mathcal{E})$ is then continuously updated (*[graph network updates]*) with the other SDN controllers. BalCon, using the updated information in the *local graph*, computes the computational load and the migration cluster in case of congestion through the *BalCon Algorithm* entity. In case of migration *BalCon Algorithm* informs *Migration Manager* entity for the local migrations and other controllers for the other migrations.

BalConController extends RYU functionalities, by supporting the multicontroller features: it can run on multiple instances on different hosts/networks (each controller has an IP address) and each instance manages a portion of the entire network. It also implements a homemade inter-controller

messaging through UDP sockets and a custom application protocol in Python. The inter-controller messaging permits the controllers to exchange themselves different kind of information like among *Graph Network Manager* entities (e.g., traffic updates) and *Migration Manager* entities (e.g., switches to be migrated). A more reliable solution could be the use of distributed data store like Zookeeper or Hazelcast [10]. *Migration Manager* module implements the switch migration procedure proposed in [9] that guarantees liveness and safety for each switch migration. Finally, BalConController fully implements the BalCon algorithm that can run indipendently in each SDN controller based on the unified view of the entire network continuously updated.

B. Functionality Test

We consider the topology in Figure 1 with a different traffic configuration: Host H_1 generates 24 new $flows/second$ to H_2, routed through S_1, S_2; host H_3 generates 30 new $flows/second$ to H_8, routed through S_3, S_4, S_7, S_8. When C_1 load reaches the threshold (here set to 10) it triggers BalCon and S_3 and S_4 are migrated to C_2. We used 4 physical machines depicted in grey (PC1, PC2, PC3, and PC4) in Figure 10. PC1 contains the hosts H_1 to H_8 implemented as Network Namespaces [2]. PC2 contains the switches S_1 to S_8 implemented using Open vSwitch v2.0.2 [3]. In order to implement different logical connections, i.e., H_1–S_1, H_2–S_2, H_3–S_3, H_6–S_6, H_8–S_8, on a single shared physical cable connecting PC1 and PC2, we configured two bridges with VLAN. PC3 and PC4 contain the two instances of BalConController, respectively. The traffic is generated using $iperf$ tool.

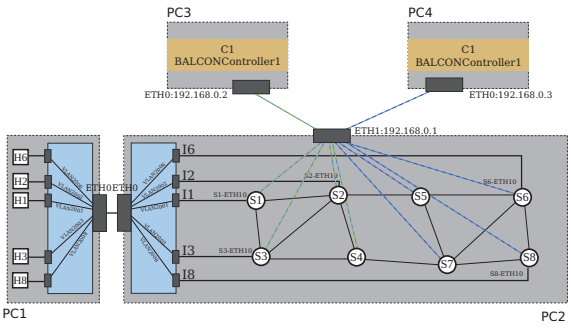

Figure 10. Functionality Test Testbed.

BalConController computes its congestion using the formula expressed in Equation 9 normalized to 10. The congestion is checked every 0.2 seconds. The load is put in an array with size 50 items, so each controller has an history of 10 seconds. If the controller surpasses the threshold more than 10 times over 50, it triggers BalCon.

Figure 11 shows the load of the two controllers and the effect of the migration of S_3 and S_4 to C_2.

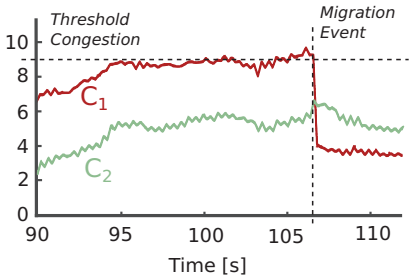

Figure 11. Functionality Test Performance.

VII. Related Works

[1], [7], [11], [26] propose multi-threaded design and parallelization techniques of OS processes in the SDN controller. [20] proposes a rethinking of the design of the SDN controllers into a lower level software that leverages both operating system optimizations and modern hardware features. [23] mitigates the scalability problem of the SDN controller by offloading all the packet inspection and creation to the GPU. Other works have also explored the implementation of distributed controllers through the using of multiple hosts: with different roles [8], [13], [27] or with equal roles [18], [19], [25]. The main focus of these papers is to address the state consistency issue across distributed controller instances, while preserving good performance. Whereas [14]–[16] focus on the controller placement problem minimizing the communication delay between controllers and switches. Current existing distributed controller solutions still suffer from the static mapping between SDN switches and controllers, limiting the capability of dynamic load adaptation. [9], [10] proposes an elastic distributed controller architecture able to force migration of SDN switches to different controllers using the existing OpenFlow standard, whereas [5] tries to model the problem of switch-controller assignment, minimizing the communication cost (in terms of hops) among controllers and switches.

VIII. Conclusion and Future Works

BalCon is a SDN switch migration mechanism able to achieve load balance among SDN controllers with small migration cost. Performance analysis shows that BalCon is effective and practical with low computational complexity and migration cost. The evaluation we did was almost entirely based on Matlab simulations with synthetic workloads. Our next objective will be: a depth analysis of BalCon in a realistic testbed; a more detail analysis on migration performances (cost and reasonable frequency); and a tradeoff analysis between a solution that requires high number of migrations but a low computation time (e.g., METIS) and BalCon that requires less migrations with an higher computation time.

REFERENCES

[1] Floodlight openflow controller. http://www.projectfloodlight.org/floodlight/. Accessed: 2014-04-24.

[2] Network namespace. http://man7.org/linux/man-pages/man8/ip-netns.8.html. Accessed: 2015-09-24.

[3] Open vSwitch. http://openvswitch.org/.

[4] RYU controller. https://osrg.github.io/ryu/.

[5] M. Bari, A. Roy, S. Chowdhury, Q. Zhang, M. Zhani, R. Ahmed, and R. Boutaba. Dynamic controller provisioning in software defined networks. In *Network and Service Management (CNSM), 2013 9th International Conference on*, pages 18–25, Oct 2013.

[6] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[7] Z. Cai, A. L. Cox, and T. S. E. Ng. Maestro: A system for scalable openflow control. Technical Report TR10-11, CS Department, Rice University, Houston, TX, USA, dec 2010.

[8] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: Scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 254–265, New York, NY, USA, 2011. ACM.

[9] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Towards an elastic distributed sdn controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 7–12, New York, NY, USA, 2013. ACM.

[10] A. A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Elasticon: An elastic distributed sdn controller. In *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '14, pages 17–28, 2014.

[11] D. Erickson. The beacon openflow controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 13–18, New York, NY, USA, 2013. ACM.

[12] M. Garey, D. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. pages 47–63. ACM, 1974.

[13] S. Hassas Yeganeh and Y. Ganjali. Kandoo: A framework for efficient and scalable offloading of control applications. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 19–24, New York, NY, USA, 2012. ACM.

[14] B. Heller, R. Sherwood, and N. McKeown. The controller placement problem. *SIGCOMM Comput. Commun. Rev.*, 42(4):473–478, Sept. 2012.

[15] D. Hock, M. Hartmann, S. Gebert, M. Jarschel, T. Zinner, and P. Tran-Gia. Pareto-optimal resilient controller placement in sdn-based core networks. In *Teletraffic Congress (ITC), 2013 25th International*, pages 1–9, Sept 2013.

[16] Y. Jimenez, C. Cervello-Pastor, and A. Garcia. Defining a network management architecture. In *Network Protocols (ICNP), 2013 21st IEEE International Conference on*, pages 1–3, Oct 2013.

[17] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.

[18] T. Koponen and et. al. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[19] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann. Logically centralized?: State distribution trade-offs in software defined networks. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 1–6, New York, NY, USA, 2012. ACM.

[20] S. Mallon, V. Gramoli, and G. Jourjon. Are today's sdn controllers ready for primetime? In *2016 IEEE 41st Conference on Local Computer Networks (LCN)*, pages 325–332, Nov 2016.

[21] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[22] Open Networking Foundation. *OpenFlow Switch Specification*, Mar. 2014. Ver. 1.3.4.

[23] E. G. Renart, E. Z. Zhang, and B. Nath. Towards a gpu sdn controller. In *2015 International Conference and Workshops on Networked Systems (NetSys)*, pages 1–5, March 2015.

[24] S. E. Schaeffer. Survey: Graph clustering. *Comput. Sci. Rev.*, 1(1):27–64, Aug. 2007.

[25] A. Tootoonchian and Y. Ganjali. Hyperflow: A distributed control plane for openflow. In *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*, INM/WREN'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[26] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood. On controller performance in software-defined networks. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[27] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with difane. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 351–362, New York, NY, USA, 2010. ACM.