Minimal Fault-tolerant Coverage of Controllers in IaaS Datacenters

Junjie Xie, *Student Member, IEEE*, Deke Guo, *Member, IEEE*, Xiaomin Zhu, *Member, IEEE*, Bangbang Ren, Honghui Chen

Abstract—Large-scale datacenters are the key infrastructures of cloud computing. Inside a datacenter, a large number of servers are interconnected using a specific datacenter network to deliver the infrastructure as a service (IaaS) for tenants. To realize novel cloud applications like the network virtualization and network isolation among tenants, the principle of software-defined network (SDN) has been applied to datacenters. In the setting, multiple distributed controllers are deployed to offer a control plane over the entire datacenter to efficiently manage the network usage. Despite such efforts, cloud datacenters, however, still lack a scalable and resilient control plane. Consequently, this paper systematically studies the coverage problem of controllers, which means to cover all network devices using the least number of controllers. More precisely, we tackle this essential problem from three aspects, including the minimal coverage, the minimal fault-tolerant coverage, and the minimal communication overhead among controllers. After modelling and analyzing such three problems, we design efficient approaches to approximate the optimal solution, respectively. Extensive evaluation results indicate that our approaches can significantly save the number of required controllers, improve the fault-tolerant capability of the control plane and reduce the communication overhead of state synchronization among controllers. The design methodologies proposed in this paper can be applied to cloud datacenters with other networking structures after minimal modifications.

Index Terms—Cloud datacenters, Minimal coverage, Fault-tolerant, Cloud computing.

1 Introduction

Large-scale datacenters are the key infrastructures for not only cloud services but also massively distributed computing. Infrastructure as a Service (IaaS) is a form of cloud computing, which enables tenants to multiplex computing, storage and network resources in data centers. With the rapid growth of IaaS, service providers have to rent or build large-scale data centers. Inside a cloud datacenter, a large number of servers are interconnected using a specific datacenter networking topology. The research community has studied various ways to improve the performance of IaaS, which can be classified into two categories.

First, to improve the network capacity inside a cloud datacenter, many novel topologies have been proposed to improve traditional tree-like topologies, such as VL2 [1], BCube [2], KCube [3], DCube [4], FBFLY [5], HyperX [6] and Jellyfish [7]. They organize all of the switches into other topologies instead of tree-like ones. Second, to realize novel cloud applications like the network virtualization and network isolation among tenants, IaaS prefers to utilize the technology of software-defined network (SDN) [8], [9] inside cloud datacenters. That is, IaaS requires to upgrade the traditional datacenter networks as software-defined datacenter networks (SDDN) [10], [11]. SDDN will change the way tenants deploy cloud-based applications since it gives the freedom to refactor the network control plane and promote the innovations of network applications. One core benefit of SDN is that the control plane is decoupled from the data plane [12], [13], [14], [15]. Accordingly, the control logic and applications are moved to a programmable software component, i.e., the controller, which maintains a global network view.

When the implementation of SDDN relies on a single controller to offer the centralized control plane, the controller is deployed on a given server inside a datacenter. Existing work has reported that one controller suffers the limitations of capacity, reliability, and scalability [16], [17]. One controller can only control a limited number of switches in large-scale datacenters. For each of uncontrolled switches, all flows passed through it will fail to obtain the routing paths from the control plane. Therefore, it is essential to deploy multiple distributed controllers in SDDN because of the super-large scale of datacenters. There have been designers who resort to deploy multiple controllers and form a physically distributed control plane, on which a logically centralized control plane operates [14], [16], [18], [19]. This, however, brings a crucial but open problem, which we call the coverage problem of controllers in SDDN.

SDDN is closely related to the networking topology of datacenters. Recently, Jellyfish is an instance of the switch-centric topology [7]. It organizes homogeneous high-radix switches, each connecting several servers, into the random regular graph. The Jellyfish topology has many distinguished performances, e.g., the incremental expansion, the cost-efficient, and the low network diameter. It, however, suffers the complex routing and poor management of network resource, due to the random topology. It is well-known that the principles of SDN can naturally address the routing and network management problems, with the support of global network view [20]. Thus, this paper employs the

- *Junjie Xie, Deke Guo, Xiaomin Zhu, Bangbang Ren and Honghui Chen are with the Science and Technology Laboratory on Information Systems Engineering, National University of Defense Technology, Changsha Hunan, 410073, China. E-mail: {xiejunjie06, guodeke}@gmail.com, xmzhu@nudt.edu.cn.*

Jellyfish as an instance of SDDN and studies the coverage problem of controllers.

In this paper, we first characterize the coverage model of a single controller and model the minimal coverage problem of controllers in SDDN. The goal is to find the minimal number of sufficient controllers and their deployment locations such that each switch can be controlled by at least one controller. We prove that this problem is NP-complete and thus propose an approximation algorithm under the Jellyfish topology. Note that more controllers will not only incur considerable cost but also complicate the synchronization process among controllers. Accordingly, this will increase the complexity of the distributed control plane of a SDDN.

Another challenge comes from common failures in datacenters [21]. Any failure due to the host server, the involved switch, and the link from the host to the switch will make a controller become invalid. Consequently, those covered switches fail to process upcoming new flows. To accommodate the failures of controllers, we propose the minimal fault-tolerant coverage problem of controllers in SDDN, a NP-complete problem. Its goal is to infer the minimal number of controllers and their locations such that each switch is controlled by at least two controllers. In this way, each switch will be taken over by another controller when the master one fails. To tackle this NP-complete problem, we further design an approximation algorithm under the Jellyfish topology.

Moreover, the distributed control plane, based on multiple controllers, operates on a consistent global network view, so as to ensure correct control behaviors [22]. This requires controllers to frequently synchronize their local network views. Such a behavior consequently results in considerable communication overhead on those limited control channels inside datacenters [23]. It further brings non-trivial synchronization delay, which can lead to suboptimal control decisions. Thus, it is essential to reduce the communication overhead of synchronization. To tackle this issue, we model and solve the minimal communication overhead of synchronization problem. Traditionally, synchronization among n controllers incurs n^2 unicast transmissions. We replace such unicast transmissions with n one-to-many multicast transmissions. Multicast refers to the minimal Steiner tree problem [24], which is a NP-hard problem in a Jellyfish datacenter. Therefore, we design a dedicated algorithm to significantly reduce the communication overload, due to the state synchronization among controllers.

The evaluation results demonstrate that our approaches significantly decrease the number of required controllers, improve the fault-tolerant capability of the control plane and reduce the communication overhead of state synchronization among controllers. Note that the proposed approaches in this paper can be applied to other switch-centric datacenter networks after some modifications.

The rest of this paper is organized as follows. In Section 2, we characterize the minimal coverage problem of controllers. Section 3 studies the fault-tolerant coverage problem of controllers. We tackle the minimal communication overhead of synchronization among controllers in Section 4. In Section 5, we evaluate the performance of our approaches. Section 6 discusses the related work. We conclude this paper in Section 7.

2 Minimal Coverage Problem of Controllers in Software-defined Datacenter Networks

We start with the coverage model of a single controller in SDDN with the Jellyfish topology. We then propose the minimal coverage problem of controllers and design a dedicated approach to approximate the optimal solution.

2.1 Coverage model of a controller in datacenters

In Jellyfish [7] topology, all of ToR switches form a random regular graph. Each ToR switch utilizes γ of κ ports, $0<\gamma<\kappa$, to interconnect with other switches. Other $\kappa-\gamma$ available ports at each switch are used to directly connect servers. Fig. 1(a) gives an example of a Jellyfish with 12 switches, each of which allocates 3 ports to connect other switches. A random regular graph is denoted as $\mathrm{RRG}(N, \kappa, \gamma)$, where N is the number of switches.

On the one hand, it is the distributed nature of networking devices that leads to complex routing and management in datacenters with Jellyfish topology. On the other hand, SDN controllers provide a global view to manage networks and make forwarding policies. By the SDN deployment, it is possible to customize the routing policy and control the flow transmission. Meanwhile, the SDN is also helpful to promote the innovation of applications in datacenters. Therefore, deploying SDN controllers in a Jellyfish-based datacenter is very useful to resolve the complex routing and network management. Before discussing the coverage problem of a controller, we first give two definitions as follows.

Definition 1. (Dominant Switch) Since the controller covers and serves some switches, the switch connecting the host server of that controller is called a dominant switch. To ease the presentation, we say that the dominant switch covers those switches served by the controller.

Definition 2. (r-cover) If the distance from a switch to a dominant switch is r hops, and the dominant switch covers such a switch. We can say that the dominant switch can r-cover the switch.

The precondition of enabling SDN functions in datacenters is that each of switches is assigned at least one controller. It is impractical to control all of switches in a large-scale datacenter, using only one controller due to the limitation of its capacity [16]. Thus, one controller is able to control a part of all switches in datacenters. If a controller controls too many switches, it will become the bottleneck of network applications. That is, the capacity of a controller determines the number of switches it can control.

Besides the capacity of a controller, the propagation latency also limits the number of switches the controller can control [25]. The propagation latency is determined by the distance from a general switch to the dominant switch of a controller. Given a threshold r, a switch r-hops away from the dominant switch of a controller has the chance to be controlled by the controller if the controller's capacity is still sufficient. In this case, we say that the dominant switch can r-cover the switch, as shown in Definition 2. If the latency constraint is one hop and its capacity exceeds four,

the dominant $switch_1$ can only 1-cover $switch_1$, $switch_2$, $switch_3$, and $switch_9$, as shown in Fig. 1(b). That is, the propagation latency and the controller's capacity jointly limit the number of switches a controller can control.

2.2 Minimal coverage problem of controllers

According to the coverage model of a controller, it is clear that SDDN should offer a distributed control plane consisting of multiple controllers. In this setting, the principal challenge is to determine the minimal number of sufficient controllers and their locations, such that each switch is controlled by at least one controller. This challenge is defined as the minimal coverage problem of controllers in SDDN.

Definition 1 demonstrates that a dominant switch covers all switches which are controlled by the related controller, including itself. Thus, the minimal coverage problem of controllers can be transferred to find the minimal set of dominant switches. For example, $\{switch_1, switch_4, switch_6, switch_7, switch_{10}\}$ is the minimal set of dominant switches in Fig. 1(b). They can 1-cover all switches in the datacenter. Given the minimal set of dominant switches, the deployed locations of controllers can be achieved in a simple way. Each dominant switch assigns one of connected servers as the host server for one controller. It is clear that some switches will not be covered if any one leaves from the minimal set of dominant switches.

So far, we can formulate the minimal coverage problem of controllers in a switch-centric datacenter as follows. We use N to denote the set of switches in a datacenter, and E to represent the set of links among switches. The topology of switches is modeled as an undirected graph, $G(N, E)$. We use a binary vector $X{=}\langle x_1, x_2, \ldots, x_{|N|}\rangle$ to denote whether a switch is a dominant switch. A switch j for $1{\leq}j{\leq}|N|$ is a dominant one only if $x_j{=}1$; otherwise, it is not a dominant switch. We further define $Y{=}[y_{ij}]_{|N|\times|N|}$, where y_{ij} is a binary variable and $y_{ij}{=}1$ means that switch j is a dominant switch and covers switch i. Let $T{=}[t_{im}]_{|N|\times|N|}$ be a traffic matrix, where t_{im} denotes the average number of flows from switch i to switch m in one second.

Given the capacity of each controller, let k denote the maximum number of flows each controller can process per second. The delay matrix is denoted as $D{=}[d_{ij}]_{|N|\times|N|}$, where d_{ij} is the propagation latency from switch i to switch j. We assume that the allowed propagation latency from switch i to a dominant switch j is ε.

The optimization objective of the coverage problem is to minimize the number of controllers, i.e., the number of dominant switches, and can be expressed as follows:

$$\min \sum_{j\in N} x_j \tag{1}$$

Meanwhile, to ensure a feasible solution, the following constraints have to be satisfied.

$$\forall j \in N: \sum_{i\in N}\sum_{m\in N} y_{ij}t_{im} \leq k \tag{2}$$

$$\forall i \in N, j \in N: y_{ij}d_{ij} \leq \varepsilon \tag{3}$$

$$\forall i \in N: \sum_{j\in N} y_{ij} \geq 1 \tag{4}$$

$$\forall i \in N, j \in N: y_{ij} \leq x_j \tag{5}$$

$$\forall j \in N: x_j \in \{0,1\} \tag{6}$$

$$\forall i \in N, j \in N: y_{ij} \in \{0,1\} \tag{7}$$

Inequality (2) guarantees that each controller can process all of flows, coming from switches controlled by it. Inequality (3) ensures that the propagation latency, between a switch and a dominant switch, is less than the given threshold. Inequalities (4) and (5) ensure that each switch is controlled by more than one controller. Equations (6) and (7) present two binary variables, x_j and y_{ij}.

The above optimization problem is an Integer Linear Programming (ILP) problem. Meanwhile, the minimal coverage problem in SDDN can be reduced to the minimum dominating set problem, and Johnson et al. have proved that the minimum dominating set problem is a NP-complete problem [26]. Furthermore, Theorem 1 shows that the minimal coverage problem is a NP-complete problem. Thus, we design a dedicated approach to approximate the optimal solution for this problem in Section 2.3.

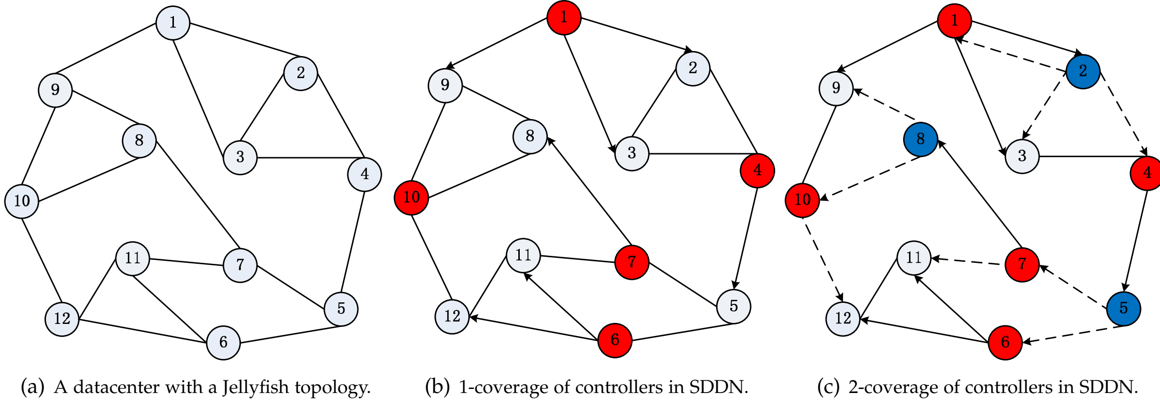
Theorem 1. The minimal coverage problem is a NP-complete problem.

Proof: A dominating set is a set of nodes S such that every node in the graph G is a neighbor of at least one node of S. The minimum domination set problem (MDS) is to find a minimum dominating set that can dominate all nodes in the graph G, and this problem is a classical NP-complete problem [26]. In fact, we can describe a polynomial reduction from MDS to a very special minimal coverage problem (MCP*), where the propagation delay is 1-hop and the controller's capacity is equal to the maximum degree of node $\Delta(G)$ in the graph G. This $\Delta(G)$ capacity ensures all switches connected to dominant switch can be controlled by the controller. In this case, an instance of MDS is also an instance of MCP*. The node in dominating set of MDS can be seen as a dominant switch in MCP*. Nodes dominated in MDS can be 1-hop away from the dominant switch. Therefore, we have shown that MDS$\leq_p$MCP*, and MCP* is NP-complete. Furthermore, it is always good to prove the NP-completeness of an especially restricted variant of a problem, since then the NP-completeness of all generalizations follows immediately. Thus the MCP is also a NP-complete problem. □

2.3 Solution to minimal coverage problem

The minimal coverage problem in SDDN differs from the traditional domination problem [27] and the coverage problem [26]. The domination problem means that any node in the dominant set can only dominate its 1-hop neighbors. The traditional coverage problem means that the node can cover its edges. For the minimal coverage problem in this paper, a dominant switch may cover other far-away switches in its coverage range, besides those 1-hop neighboring switches. The coverage range of each controller may be more than one hop. Moreover, a dominant switch can not cover a switch if its capacity is insufficient, even the constraint on the propagation latency is satisfied. That is, the minimal coverage problem in this paper suffers extra constraints and is more complex than the traditional domination problem and the coverage problem. Thus, it can not be well addressed by simply using existing solutions to such two problems.

Therefore, we propose an approximation algorithm based on the complete control, denoted as ACC. The input



(a) A datacenter with a Jellyfish topology. (b) 1-coverage of controllers in SDDN. (c) 2-coverage of controllers in SDDN.

Fig. 1. The 1-coverage and 2-coverage of controllers in a software-defined datacenter with the Jellyfish topology.

of the algorithm ACC includes a Jellyfish topology $G(N, E)$, the number of hops that can meet the time constraint r, and the capacity of each controller u. Note that the capacity of a controller is measured by the number of switches it can control. The time constraint for each flow is measured by the number of hops from the original switch to a related controller.

As shown in Algorithm 1, we first derive the neighboring switches of each switch from the datacenter's topology G. We then select a switch that can cover the largest number of new switches as a dominant switch, and we achieve this by the for loop in Step 6. The dominant switch hosts a controller and tries to cover a set of switches recursively, by invoking the function $CONTROL()$ in Algorithm 1. The above steps are executed repeatedly until all switches are covered by a controller. That is, Algorithm 1 will terminate when all switches are covered. Meanwhile, the function $CONTROL()$ performs the breadth-first search. The search process will be terminated if the depth of recursion exceeds allowed hops or the number of covered switches exceeds the controller's capacity. In each round of recursion, all of num-hop neighbor switches of the dominant switch will be checked orderly, where num denotes the current depth of the recursion. If a num-hop neighbor switch has not been controlled, the dominant switch will cover it, and the available capacity of the controller decreases one. The function $getControlNum()$ is similar to the function $CONTROL()$ and used to get the number of switches covered by $switch_i$. Algorithm 1 can ensure that each switch in a datacenter will be controlled by at least one controller, i.e., covered by at least one dominant switch. Theorem 2 shows that the approximation ratio of the Algorithm 1 is lower than $1 + ln(u)$.

Theorem 2. The approximation ratio of the algorithm 1 is lower than $1 + ln(u)$, where u denotes the capacity of a controller.

Proof: Let A^* denote the optimal solution. A is the approximation solution the Algorithm ACC generated. Suppose that the elements of A include $x_1, x_2, \ldots, x_k$, and they are orderly added to A. Let A_i=$\{x_1, x_2, ..., x_i\}$ where x_i is a dominant switch; specially, A_0=$\emptyset$. Let S denote the set which includes all switches. For each $B \subseteq S$, use $c(B)$ to denote the number of the dominant switches in B. We use $H(u)$ to denote the harmonic function $H(u) = \sum_{i=1}^{u} \frac{1}{i}$. Note that $H(u) \leq 1 + ln(u)$.

ALGORITHM 1: Find the minimal set of controllers, ACC

```
   Input: G(N,E), r, and u.
   Output: Set of controllers, C
1  Let CS[N] record if the switches has been covered;
2  Let Nbors record the relationships between switches;
3  Construct Nbors based on the topology G;
4  while there are still switches that are not covered do
5  |  maxi=0, maxnum=0;
6  |  for i=1 to N do
7  |  |  if switch_i is not controlled then
8  |  |  |  tempnum=getControlNum(switch_i);
9  |  |  |  if tempnum ≥ maxnum then
10 |  |  |  |  maxi=i;
11 |  |  |  |  maxnum=tempnum;
12 |  switch_maxi is selected as a dominant switch;
13 |  Let num=1 denote the control distance;
14 |  CS ← CONTROL(CS, Nbors, r, u, maxi, num);
15 return the set of controllers, C;

16 function CONTROL(CS, Nbors, r, u, i, num)
17 |  if num > r or exceed the capacity of the controller
   |  then
18 |  |  return CS;
19 |  else
20 |  |  for j=1 to i.neighbor.size() do
21 |  |  |  if switch_j has not been controlled then
22 |  |  |  |  switch_j is controlled by the controller;
23 |  |  |  |  Controller's available capacity
   |  |  |  |  decreases one;
24 |  |  |  |  if the controller exhaust its capacity then
25 |  |  |  |  |  break;
26 |  |  for j=1 to i.neighbor.size() do
27 |  |  |  tempj = i.neighbor[j];
28 |  |  |  num = num + 1;
29 |  |  |  CONTROL(CS, Nbors, r, u, tempj, num);
30 |  |  return CS ;
```

To simplify the notation, use r_i= $\triangle_{x_i} f(A_{i-1})$ to denote the number of new switches the $switch_{x_i}$ can cover, when these switches in A_{i-1} have coved some switches. Let $z_{y,i}$= $\triangle_y f(A_{i-1})$. For each element y in A^*, define

$$w(y) \equiv \sum_{i=1}^{k}(z_{y,i} - z_{y,i+1})\frac{1}{r_i} = \frac{1}{r_1}z_{y,1} + \sum_{i=2}^{k}(\frac{1}{r_i} - \frac{1}{r_{i-1}})z_{y,i}.$$

Similarly,

$$\begin{aligned} c(A) &= \sum_{i=1}^{k} \frac{r_i}{r_i} = \sum_{i=1}^{k} (\sum_{j=i}^{k} r_j - \sum_{j=i+1}^{k} r_j) \frac{1}{r_i} \\ &= \frac{1}{r1} \sum_{j=1}^{k} r_j + \sum_{i=2}^{k} (\frac{1}{r_i} - \frac{1}{r_{i-1}}) \sum_{j=i}^{k} r_j. \end{aligned}$$

According to the greedy policy of the algorithm 1, $r_1 \geq r_2 \geq \cdots \geq r_k$; equally, for each i=$1, 2, \ldots, k$, $\frac{1}{r_i} - \frac{1}{r_{i-1}} \geq 0$.

Moreover,

$$\begin{aligned} \sum_{j=i}^{k} r_j &= \sum_{j=i}^{k} \triangle_{x_j} f(A_{j-1}) \\ &= \sum_{j=i}^{k} (f(A_j) - f(A_{j-1})) = f(A) - f(A_{i-1}) \\ &= f(A^*) - f(A_{i-1}) = \triangle_{A^*} f(A_{i-1}) \\ &\leq \sum_{y \in A^*} \triangle_y f(A_{i-1}) = \sum_{y \in A^*} z_{y,i}. \end{aligned}$$

So, $\sum_{j=i}^{k} r_j \leq \sum_{y \in A^*} z_{y,i}$.

Then, the inequality (8) holds.

$$\begin{aligned} c(A) &= \frac{1}{r1} \sum_{\mathrm{j=1}}^{\mathrm{k}} r_j + \sum_{\mathrm{i=2}}^{\mathrm{k}} (\frac{1}{r_i} - \frac{1}{r_{i-1}}) \sum_{\mathrm{j=i}}^{\mathrm{k}} r_j \\ &\leq \sum_{\mathrm{y \in A^*}} \frac{1}{r_1} z_{y,1} + \sum_{\mathrm{y \in A^*}} \sum_{\mathrm{i=2}}^{\mathrm{k}} (\frac{1}{r_i} - \frac{1}{r_{i-1}}) z_{y,i} = \sum_{\mathrm{y \in A^*}} w(y) \end{aligned} \tag{8}$$

Due to the greedy policy of the algorithm 1, when $z_{y,i} \geq 0$, for each i=$1, 2, \ldots, k$, $\frac{1}{r_i} \leq \frac{1}{z_{y,i}}$. Meanwhile, $z_{y,i} \geq z_{y,i+1}$. Let ℓ=$max\{i | 1 \leq i \leq k, z_{y,i} \geq 0\}$,

$$\begin{aligned} w(y) &= \sum_{i=1}^{\ell} (z_{y,i} - z_{y,i+1}) \frac{1}{r_i} \leq \sum_{i=1}^{\ell} (z_{y,i} - z_{y,i+1}) \frac{1}{z_{y,i}} \\ &= \sum_{i=1}^{\ell} \frac{z_{y,i} - z_{y,i+1}}{z_{y,i}}. \end{aligned}$$

Note that for any $p \geq q \geq 0$,

$$\frac{p-q}{p} = \sum_{j=q+1}^{p} \frac{1}{p} \leq \sum_{j=q+1}^{p} \frac{1}{j} = H(p) - H(q).$$

Therefore,

$$w(y) \leq \sum_{i=1}^{\ell-1} (H(z_{y,i}) - H(zy, i+1)) + H(z_{y,\ell}) = H(z_{y,1}).$$

For each y in A^*, $z_{y,1} \leq u$. Then, there is the equation (9).

$$w(y) \leq H(u) \tag{9}$$

Based on the equation (8) and (9), we can get $c(A) \leq \sum_{y \in A^*} w(y) \leq \sum_{y \in A^*} H(u) = c(A^*) \cdot H(u) \leq c(A^*) \cdot (1 + ln(u))$. Thus proved. □

For Algorithm 1, let h denote the maximum number of switches a controller actually control in the software-defined datacenter. The time complexity of the function Control is $O(h)$. Let ℓ denote the number of controllers that are solved by Algorithm 1. The main function has one while loop, and its time complexity is $O(\ell)$. The time complexity of the for loop is $O(N)$, where N is the number of switches in the datacenter. Therefore, The time complexity of Algorithm 1 is $O(N \times h \times \ell)$.

3 Fault-tolerant Coverage Problem of Controllers in Software-defined Datacenters

In this section, we consider the fault-tolerant requirement of controllers. Although the minimal coverage problem of controllers has been tackled in Section 2, it is essential to guarantee that each switch still supports all of SDN functions when its controller fails.

3.1 Problem of the minimal fault-tolerant coverage

Recently, one trend of designing modern datacenters is to utilize commodity devices, including servers and switches. Such large-scale devices face various failures, resulting from the hardware, software, link, etc. As pointed in many literatures, device failure is very common in current datacenters [21]. The Onix controller has considered four types of network failures [14]: forwarding element failures, link failures, controller failures and failures in connectivity between network elements and controllers (and between the controllers themselves). Such kinds of failures would make a controller fail to serve the allocated switches. In this case, there are switches that are not covered by any controller. Consequently, this will destroy the guarantee of the minimal coverage model of controllers proposed in this paper. To tackle such a crucial issue, we propose the minimal fault-tolerant coverage model of controllers in SDDN.

The basic idea is to calculate the minimal number of controllers and their locations; thus, each switch will be controlled by at least two controllers. Accordingly, each switch will be taken over by another controller when the master one fails. We call this problem as the 2-coverage in Definition 3.

Definition 3. (2-coverage) Each switch is controlled by two controllers, i.e., covered by two dominant switches. Meanwhile, each dominant switch is covered by another dominant switch.

It is clear that more strength fault-tolerant coverage models can better against the failure of controllers. These models, however, dramatically increase the number of deployed controllers. For this reason, we focus on the 2-coverage in this paper. The proposed methodologies, however, can be applied to other fault-tolerant coverage models, such as the 3-coverage model.

Recall that we have deployed a set of controllers to ensure that each switch has been covered by at least one controller. As shown in Fig. 1(c), five controllers are deployed to locations $switch_1$, $switch_4$, $switch_6$, $switch_7$, and $switch_{10}$ so as to realize the 1-coverage problem. To address the 2-coverage problem, some extra controllers need to be further deployed. For example, three new controllers are deployed to locations $switch_2$, $switch_5$, and $switch_8$. In this way, such controllers need to guarantee that each switch can be controlled by two controllers. Any switch can be controlled by another slave controller in a cooperative manner, when its master controller fails. Meanwhile, the controllers cooperate to ensure the fault-tolerant coverage. When each switch is controlled by two controllers, the fault-tolerant coverage is achieved. For any general switch, the controller at the other end of its solid incoming arc denotes the master controller, and the controller at the end of its dotted incoming arc is the slave controller. For a general switch $switch_9$ in Fig. 1(c), its master and slave controllers are attached to $switch_1$ and $switch_8$, respectively.

While solving the 2-coverage problem, the number of utilized extra controllers should also be as fewer as possible. We model this problem as the minimal 2-coverage problem in Definition 4.

Definition 4. (Minimal 2-coverage) The minimal 2-coverage problem means to find the minimal set of controllers to satisfy the 2-coverage in Definition 3.

We formulate the minimal 2-coverage problem in SDDN as an optimization problem as follows. Note that prior Inequality (4) imposes a constraint that each switch is controlled by at least one controller. However, for the 2-coverage problem, each switch should be controlled by at least two controllers. Thus, Inequality (4) should be updated as Inequality (10) in this setting. Additionally, Equations (2), (3), (5), (6), and (7) are inherited. We introduce another binary vector $C = \langle c_1, c_2, \ldots, c_{|N|} \rangle$, where $c_j = 1$ for $1 \leq j \leq |N|$ means that a controller has been deployed to a host connected to switch j via Algorithm 1. Equation (11) introduces a new constraint to ensure that those controllers, determined by Algorithm 1, continue to be valid.

$$\forall i \in N : \sum_{j \in N} y_{ij} \geq 2 \tag{10}$$

$$\forall j \in N : x_j \geq c_j \tag{11}$$

The optimization objective is to minimize the number of additional controllers such that each switch is controlled by two controllers, and can be expressed as follows:

$$\min \sum_{j \in N} (x_j - c_j) \tag{12}$$

The optimization problem is an Integer Linear Programming (ILP) problem and is similar to the minimal coverage problem of controllers in Section 2.2. Corollary 1 shows that the minimal 2-coverage problem is also a NP-complete problem and very difficult to find the optimal solution in polynomial time. We then propose a dedicated algorithm to approximate the optimal solution.

Corollary 1. The minimal 2-coverage problem in software-defined datacenters is NP-complete.

Proof: The minimal 2-coverage problem is to use the minimal number of controllers while ensuring that each switch is controlled by two controllers. That is, find the minimal number of dominant switches to ensure that each switch is covered by two dominant switches. The minimal coverage problem is to find the minimal number of controllers while ensuring that each switch is controlled by one controller. Because Theorem 1 shows that the minimal coverage problem in software-defined datacenters is NP-complete, the minimal 2-coverage problem is also NP-complete. Thus proved. □

3.2 Solution to fault-tolerant coverage problem

To achieve the minimal 2-coverage, we propose an approximation Algorithm 2, denoted as ABC. The set of controllers resulting from Algorithm 1 acts as one of the input. Let $CS[N]$ record if the switches has been covered by two controllers. Algorithm 2 firstly derives the relationships of switches, based on the datacenter topology G. It then checks the state of each switch in the datacenter in turn. Since each switch only needs to be controlled by one controller in Algorithm 1, some controllers might have not been fully utilized. If a switch has been a dominant switch and its controller is in C, the controller will continue to control other switches when its capacity is still sufficient. Furthermore, if there exist switches, which have not been covered by two controllers, Algorithm 2 will select a switch that can cover the most number of other switches as the dominant switch, which is achieved by the for loop in Step 10. The dominant switch will host a controller. Algorithm 2 will terminate when all switches have been controlled by at least two controllers. Similarly, the approximation ratio of Algorithm 2 is also lower than $1 + ln(u)$, and u denotes the capacity of a controller.

ALGORITHM 2: Finding extra controllers, ABC

Input: $G(N, E)$, r, and u; Let C denote a set of controllers resulting from Algorithm 1.
Output: the set of extra controllers, $C2$.

1 Let $CS[N]$ record if the switches has been covered by two controllers;
2 Let $Nbors$ record the relationships between switches;
3 Construct $Nbors$ based on the topology G;
4 Construct $CS[N]$ based the set of controllers C;
5 **for** i=1 *to* N **do**
6 | **if** *a controller has deployed in the location of* $switch_i$ *in Algorithm 1* **then**
7 | | Let the controller control other switches;
8 **while** *There exist other switches that are not covered by two controllers* **do**
9 | Let $maxi$=0, $maxnum$=0;
10 | **for** i=1 *to* N **do**
11 | | **if** $switch_i$ *is not a dominant switch* **then**
12 | | | $tempnum$=$getControlNum(switch_i)$;
13 | | | **if** $tempnum \geq maxnum$ **then**
14 | | | | $maxi$=i;
15 | | | | $maxnum$=$tempnum$;
16 | $switch_{maxi}$ is selected as a dominant switch;
17 | $switch_{maxi}$ is added into $C2$;
18 | Let $switch_{maxi}$ control other switches;
19 return the set of extra controllers, $C2$.

The time complexity of the first for loop in Algorithm 2 is $O(N \times h)$, where N is the number of switches in a datacenter, and h denotes the maximum number of switches a controller actually controls. Let t denote the number of controllers that are identified by Algorithm 2. Then, the time complexity of the while loop in Algorithm 2 is $O(N \times h \times t)$. Therefore, the time complexity of Algorithm 2 is $O(N \times h \times t)$.

4 MINIMAL COMMUNICATION OVERHEAD OF SYNCHRONIZATION AMONG CONTROLLERS

We start with studying the minimal communication overhead of Synchronization among controllers, after realizing the minimal and fault-tolerant coverage of controllers. Thus, we propose a dedicated approach to considerably reduce the communication overhead due to the state synchronization.

4.1 Problem description

After realizing the minimal and fault-tolerant coverage of controllers, those controllers must operate on a consistent

global network view, so as to ensure correct control behaviors. Each controller, however, only maintains a local network view, which frequently changes due to many dynamic network behaviors [22]. To realize a consistent view of the global network among controllers, Onix builds a shared network information base that is a data structure including all network entities within a network topology [14]. However, the instance of network information base has to handle the replication and distribution of data to all of controllers. Additionally, there are also some researchers who seek to achieve the benefits of network control centralization by passively synchronizing network-wide views of controllers, such as HyperFlow [18]. It localizes decision making to individual controllers and minimizes the control plane response time to data plane requests.

Traditionally, synchronization among n controllers incurs n^2 unicast. This process incurs non-trivial traffic overhead in a datacenter, and imposes the synchronization delay, which can lead to suboptimal control decisions. Thus, it is essential to minimize the synchronization overhead and delay. In this paper, we tackle this issue from two aspects. First, we deploy as few controllers as possible in SDDN, while ensuring the 2-coverage requirement. Among those controllers, only the first part of controllers, which are deployed in Section 2 need to synchronize their states. When a master controller is broken, the related slave controller will get the up-to-date states from other controllers.

Second, we replace the n^2 unicast transmissions with n one-to-many multicast transmissions among n controllers. Such n multicast transmissions share the same multicast tree. If the multicast tree utilizes the least network links, we can minimize its communication overhead. The main challenge is to find the minimal Steiner tree (MST) for such n controllers in a datacenter. It is well-known that this is a NP-hard problem in a general graph [24]. The Jellyfish topology is a random regular graph and its minimal Steiner tree is also NP-hard.

There exist many approximation methods for solving the MST problem. Although some approximation methods for the MST problem obtain the better approximation ratio, they exhibit higher computation complexity compared to the one proposed in [28]. Meanwhile, the approximation ratio of the Steiner algorithm [28] is lower than 2. The time complexity of the Steiner algorithm is $O(C \times N^2)$, where C is the number of controllers and N is the number of switches in a datacenter. Although the Steiner algorithm is fast, it cannot generate efficient multicast trees online for large number of multicast groups simultaneously, due to the increasing scale of datacenters. Therefore, there is an urgent need of an efficient and fast algorithm for the MST problem in a large datacenter. To ease the presentation, we further formulate this problem in the next section.

4.2 Problem formulation

In this section, we formulate the minimal Steiner tree in a Jellyfish datacenter as an Integer Programming (IP) problem. Let N denote the number of switches in the datacenter. Let an adjacent matrix $G = [a_{ij}]_{|N|\times|N|}$ denote the topology of a datacenter at the level of switch. a_{ij}=1 means that there is an edge between node i and node j; otherwise, a_{ij}=0. Let C denote the set of controllers, whose local network views need to be synchronized.

We use W=$[w_{ij}]_{N\times N}$ to denote a Steiner tree, where w_{ij}=1 means that an edge between node i and node j is selected to appear in the Steiner tree. Moreover, if a_{ij}=0, w_{ij} can not be 1. According to Theorem 3, W^n=$[w^n_{ij}]_{N\times N}$ is a reachable matrix with a constraint that the path length between any pair of nodes is n. Let w^n_{ij} denote the number of paths with length n from node i to node j. Let Z= $[z_{ij}]_{|N|\times|N|}$ ={I+W+W^2+W^3+ . . . +W^{N-2}+W^{N-1}} denote the reachable matrix, where I is the unit matrix, w_{ij}=1 when i=j, otherwise, w_{ij}=0. If z_{ij}>0, it means that the message of node i can reach node j. Otherwise, z_{ij}=0.

Theorem 3. W^n=$[w^n_{ij}]_{|N|\times|N|}$ represents a reachable matrix, where the path length between any pair of nodes is n.

Proof: When n=1, W^n=W=$[w_{ij}]_{|N|\times|N|}$ denotes a reachable matrix, where the path length between any pair of nodes is 1. When n=k, W^k=$[w^k_{ij}]_{|N|\times|N|}$ denotes a reachable matrix, where the path length between any pair of nodes is k. When n=k+1, W^{k+1}=$[w^{k+1}_{ij}]_{|N|\times|N|}$=$W^k\times W$= $\sum_{l\in N} w^k_{il}\times w_{lj}$. It is clear that W^{k+1} represents a reachable matrix, where the path length between any node pair is k+1. So, W^n=$[w^n_{ij}]_{|N|\times|N|}$ represents the reachable matrix, where the path length between any two nodes is n. Thus, Theorem 3 is proved. □

The optimization objective of the cost synchronization problem is to minimize the number of edges in the related Steiner tree, and can be expressed as follows:

$$\min \sum_{i,j\in N} w_{ij}. \tag{13}$$

Meanwhile, the following constraints must be satisfied to guarantee a feasible solution:

$$\forall i \in N, j \in N : w_{ij} \leq a_{ij} \tag{14}$$

$$\forall i \in C, j \in C : z_{ij} > 0 \tag{15}$$

$$\forall i \in N, j \in N : w_{ij} \in \{0, 1\} \tag{16}$$

Inequality (14) ensures that the edge must be selected from the topology of the datacenter. Inequality (15) guarantees that all synchronization controllers are selected and connected. Constraints (16) indicates that w_{ij} is a binary variable. This is an Integer Programming problem and is also a NP-hard problem. Thus, we propose a dedicated algorithm to approximate the optimal solution in the next section.

4.3 Solution to minimal communication overhead of synchronization among controllers

To construct the minimal Steiner tree, we first derive a multi-stage graph from the datacenter topology. The source node in the graph can be any controller, which needs to synchronize its state. A node in the graph may have multiple father nodes. For any node, if another node locates at the same layer and connects to it, such a node is called a brother node of the current one. Fig. 2 indicates a multi-stage graph, resulting from a Jellyfish topology in Fig. 1. The node $switch_1$ is the source node and locates at the stage 0. The node $switch_1 1$ has two father nodes, $switch_7$ and

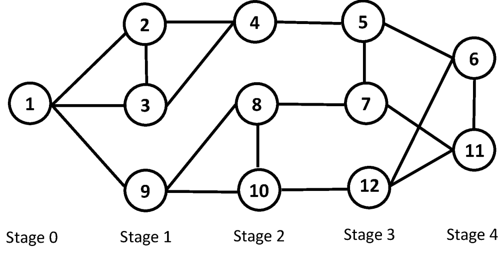


Fig. 2. A multi-stage graph from the Jellyfish topology in Fig. 1.

ALGORITHM 3: Find the Steiner tree, AMG

```
    Input: The multi-stage graph, T; the set of controllers, C.
    Output: Steiner tree, W
 1  for i=lastlevel to 1 do
 2    Get elements in the level i;
 3    for j=1 to elems.size() do
 4      flag=0;
 5      if elem[j] is a controller or elem[j] is selected as a relay node then
 6        Get the father nodes of elem[j];
 7        for x=1 to fnode.size() do
 8          if fnode[x] must be passed then
 9            Select the edge between elem[j] and fnode[x];
10            flag=1; break;
11        if flag==0 then
12          Get the brother nodes of elem[j];
13          for y=1 to bnode.size() do
14            if bnode[y] must be passed and the edge between bnode[y] and elem[j] has not been selected then
15              Select the edge between bnode[y] and elem[j];
16              flag=1; break;
17        if flag==0 then
18          Select the edge between elem[j] and a father node;
19  return the selected edges W.
```

$switch_1 2$. The nodes, $switch_2$ and $switch_8$, are the brother nodes of $switch_3$ and $switch_{10}$, respectively.

After that we construct the multi-stage graph from the datacenter topology, we need to select some switch nodes and edges to construct the minimal Steiner tree. Those dominant switches, determined in Section 2, have to be selected into the Steiner tree. We need to find those Steiner nodes to connect such dominant switches with as low cost as possible.

Definition 5. (Relay Node) The relay node is a switch node in the multicast tree, and the relay node is neither a source node nor a destination node. However, the relay node will transfer data to the destination nodes for the multicast transfer.

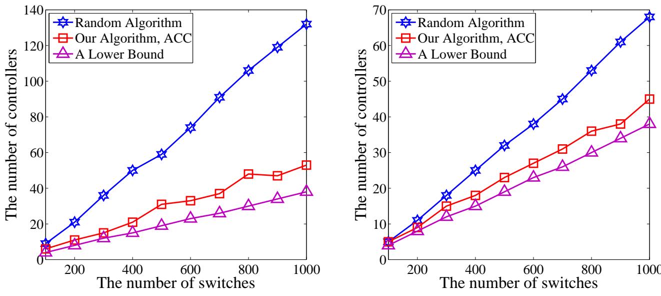
We propose an approximation algorithm 3, AMG, which constructs the Steiner tree in a bottom-up way, given the multi-stage graph. That is, the algorithm starts from all nodes at the last stage in the multi-stage graph to the nodes at stage 1. If a switch node is associated with a synchronization controller or acts as a relay node defined in Definition 5, it has to be selected. If the current switch is a relay node, Algorithm 3 first checks its father nodes in the multi-stage graph. If one of its father node is also a relay node, the edge connected to such father node should be selected. Otherwise, Algorithm 3 selects the edge between the relay switch and one of its brother nodes, if at least one brother node is a relay node. If all of its father nodes and brother nodes are not selected, Algorithm 3 randomly selects a father node, which will become a relay node. Such operations are repeated until the nodes at stage 1 is checked in the multi-stage graph.

The first two 'for' loops in Algorithm 3 try to traverse each switch in the datacenter. So the time complexity is $O(N)$, where N is the number of switches in a datacenter. The length of the first loop in Step 1 is the number of stages in the multi-stage graph. The length of the second loop in Step 3 is related to the number of nodes in each stage. The first and the second loops will ensure that all nodes are visited once. Therefore, we say that the complexity is $O(N)$ where N is the number of nodes in the multi-stage graph. The last two 'for' loops try to visit the neighboring switches of a switch. The corresponding time complexity is $O(\gamma)$, where γ denotes the number of ports per switch allocated for switch interconnections in a Jellyfish datacenter. The third loop in Step 7 and the fourth loop in Step 13 are to visit those father nodes and brother nodes, respectively. The total number of father nodes, brother n-odes and child nodes is γ. Meanwhile, the third loop and the fourth loop are parallel. Therefore, the time complexity is $O(\gamma)$. Therefore, the time complexity of Algorithm 3 is $O(N\times\gamma)$. For a Jellyfish datacenter with 1000 switches, there are 16000 servers, when each switch allocates 8 ports for switch interconnections. Owing to the time complexity of the Steiner algorithm in [28] is $O(C\times N^2)$. It is obvious that $N\times\gamma=1000\times8\leq C\times1000^2=C\times N^2$, where $C\geq1$. Therefore, Algorithm 3 is faster than the Steiner algorithm in [28].

5 Performance Evaluation

We conduct extensive simulations to evaluate the performance of our algorithms, which are used to tackle the minimal coverage, the minimal fault-tolerant coverage, and minimal communication overhead of state synchronization problems. We do these simulations based on the Jellyfish topologies, which are random regular graphs. The results show that our algorithms always exhibit good performance in these settings.

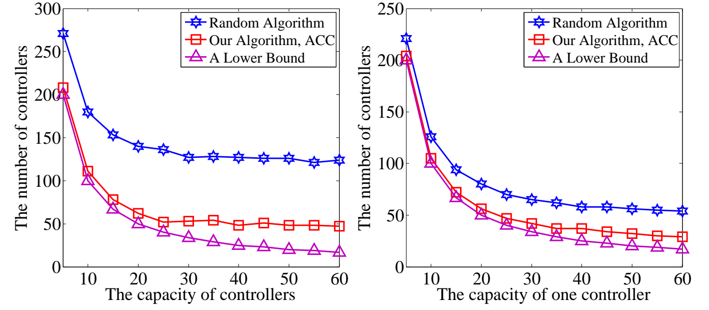
For the minimal coverage and minimal fault-tolerant coverage problems of controllers, the main input parameters include the network topology, the controller capacity and the propagation delay constrains. The random networks are abstracted as the random graph, which are constructed based on the construction method [7]. Given a network, the objective is to find the number of employed controllers and their locations. Our algorithms can be achieved by any programming language in any computing platform. Through abstracting the network topology and inputting



(a) A switch uses 8 ports for switches' interconnection.

(b) A switch uses 12 ports for switches' interconnection.

Fig. 3. The impact of the amount of switches on the number of controllers, where each switch possesses 24 ports.



(a) A switch uses 8 ports for switches' interconnection.

(b) A switch uses 12 ports for switches' interconnection.

Fig. 4. The impact of the controller's capacity on the number of controllers, where the datacenter has 1000 switches.

corresponding parameters, our algorithms can achieve the number of required controllers and their locations. For the minimal communication overhead of synchronization among controllers, the main goal is to find employed links for multicast transfer. Given the network topology and the locations of controllers, our algorithm 3 can achieve the employed links for synchronization.

5.1 Performance of the minimal coverage method

As far as we know, there exists no other appropriate method, which can be compared with our methods. Therefore, we evaluate the performance of Algorithm 1 and prior random algorithm, in the term of the number of required controllers. The impact of datacenter size and controller's capacity are reported in Section 5.1.1 and Section 5.1.2, respectively. In each setting, we construct 10 random networks. The showing results is the average value under the 10 random networks. For the random placement, we will run it 100 times to achieve the average result in each random network.

The insight behind the random algorithm is to randomly select the location for a controller, which will control around switches according to the coverage model in Section 2.1. If at least one switch have not been controlled, a new controller is randomly deployed. This algorithm repeats until each switch in a datacenter has been controlled by at least one controller.

5.1.1 Impact of datacenter size

We evaluate the performance of Algorithm 1 under variable datacenter sizes. The results indicate that it always utilizes much fewer controllers than the random algorithm, irrespective of the datacenter size.

It is clear that the capacity of a controller directly affects the number of required controllers. Kandula et al. have found that a cluster of 1500 servers exhibits a median flow arrival rate, i.e., 100K flows per second [29]. The average generation rate of flows at a server is 67 in a second. When a switch with 24 ports allocates 8 ports for interconnection with other switches, a switch can connect 16 servers. The median flow arrival rate for a switch is 1072 per second. Additionally, an individual controller can deal with at least 30K flows per second [30]. Thus, it is easy to calculate that the capacity of a controller is 27. That is, one controller can control 27 switches. The propagation delay in datacenters is mainly decided by hops while it is affected by the physical distance in wide-area networks. In Jellyfish datacenters, the number of switches that are 2 hops away from a controller is between 8 and 64 when each switch uses 8 ports for switches' interconnection. Therefore, we set 2 hops as the coverage range of each controller. It means that the farthest switch a controller can control is 2-hops away, so as to satisfy the constraint of the propagation latency.

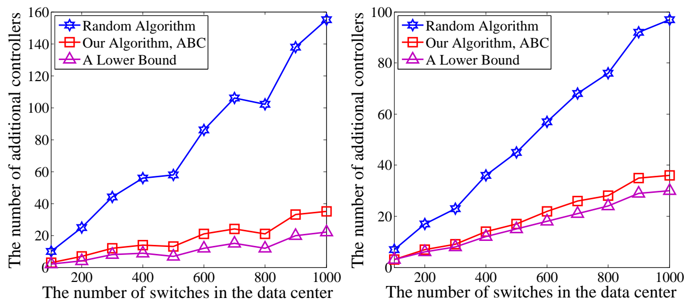
Fig. 3(a) shows that Algorithm 1 always deploys fewer controllers than the random algorithm, where each switch reserves 8 of 24 ports for connecting other switches, irrespective of the number of switches. Additionally, the number of controllers Algorithm 1 deploys is close to the lower bound. The lower bound is equal to the ratio of the total number of switches to the capacity of one controller. The coverage ranges of controllers may overlap, thus even the optimal solution is larger than the lower bound. The required number of controllers increases as the number of switches varying from 100 to 1000, i.e., the datacenter size increases from $1,600$ to $16,000$. To measure the influence of port allocation policy at each switch, we further make each switch reserve 12 of 24 ports for connecting switches. We can see from Fig. 3(b) that Algorithm 1 still outperforms the random algorithm, irrespective of the number of switches. Moreover, the number of controllers deployed by Algorithm 1 gets closer to the lower bound. The similar results are achieved, when the number of ports used to interconnect switches is set to other values.

5.1.2 Impact of controller's capacity

We evaluate the performance of Algorithm 1 under variable capacity of a controller. We find that it always utilizes much fewer controllers than the random algorithm, irrespective of the capacity of a controller.

Given a Jellyfish datacenter with 1000 switches, the capacity of each controller varies from 10 to 50. That is, the amount of switches one controller can control ranges from 10 to 50. To satisfy the constraint on the propagation latency, the coverage range of each controller is at most 2 hops. We evaluate Algorithm 1 and the random algorithm, when each switch assigns 8 or 12 ports to connect other switches.

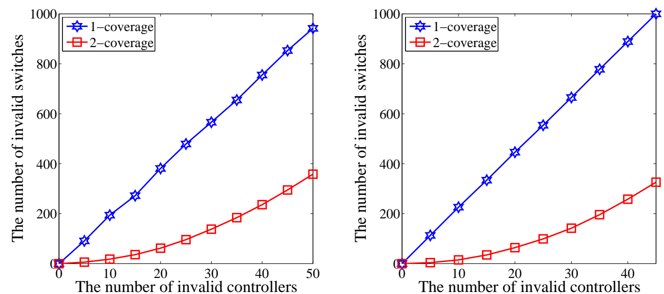
Fig. 4(a) shows that Algorithm 1 always utilizes much fewer controllers than the random algorithm, where each switch uses 8 ports for switches' interconnection. Additionally, the amount of required controllers decreases as the capacity of a control ranges from 10 to 60, under our algorithm and the random algorithm. The number of required



(a) A switch uses 8 ports for switches' interconnection.

(b) A switch uses 12 ports for switches' interconnection.

Fig. 5. The impact of the number of switches on the number of extra controllers to achieve 2-coverage.



(a) A switch uses 8 ports for switches' interconnection.

(b) A switch uses 12 ports for switches' interconnection.

Fig. 6. The impact of invalid controllers on switches in the datacenter with 1000 switches, each of which has 24 ports.

controllers, however, will not continue to decrease, when the capacity of a controller approaches and exceeds 50. The reason is that the coverage range of each controller becomes the major impact factor. Although a controller has sufficient capacity, other switches, which is too far from the controller, do not satisfy the constraint on the propagation latency. Meanwhile, in Fig. 4(a), we give the lower bound, which is equal to the ratio of the total number of switches to the capacity of one controller. The optimal solution is larger than the lower bound because the control ranges of controllers may overlap. The number of controllers Algorithm 1 deploys is close to the lower bound. We can see that the number of controllers our Algorithm 1 derives is very close to the lower bound when the capacity of one controller is not sufficient large from Fig. 4(a).

To measure the influence of port allocation policy at each switch, we make each switch allocate 12 ports for connecting switches. We can see from Fig. 4(b) that Algorithm 1 still outperforms the random algorithm, irrespective of the capacity of each switch. We found that the similar results are achieved, when the number of ports used to interconnect switches is set to other values besides 8 and 12. In a summary, Algorithm 1 always deploys fewer controllers than the random algorithm. Moreover, when the capacity of a controller exceeds a threshold, the major factors affecting the amount of controllers become the number of allocated ports for connecting switches and the constraint on the propagation delay.

5.2 Performance of the minimal fault-tolerant coverage method

In this section, we evaluate the cost and effectiveness of Algorithm 2 when addressing the minimal fault-tolerant coverage problem.

5.2.1 The number of extra controllers

Note that Algorithms 1 and 2 deal with the 1-coverage and 2-coverage problems of controllers in SDDN, respectively. The goal of Algorithm 2 is to add the minimal number of extra controllers to the set of existing controllers derived by Algorithm 1. In this section, we compare Algorithm 2 with the random algorithm under different size of datacenters, in terms of the new controllers. Given the set of controllers for addressing the 1-coverage problem, the random algorithm randomly redeploys some new controllers until the constraint on the 2-coverage problem is satisfied.

We conduct evaluations when the number of switches ranges from 100 to 1000 in datacenters. The set of controllers resulting from Algorithm 1 is one of the input of Algorithm 2. Other input variables are inherited from Section 5.1.1. After conducting 100 rounds of experiments in each setting, we calculate the average number of new controllers for the two algorithms.

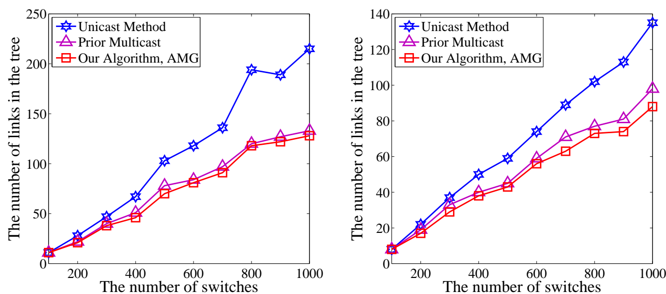
Fig. 5(a) indicates that the number of extra controllers calculated by Algorithm 2 is much fewer than the random algorithm, where each switch allocates 8 ports for switch interconnection. Moreover, when the amount of switches in a datacenter increases, the benefit of Algorithm 2 becomes more prominent. Meanwhile, we can see that the number of controllers our Algorithm 2 derives is very close to the lower bound from Fig. 5(a) and Fig. 5(b). The lower bound is equal to the difference between the ratio of the 2 times total number of switches to the capacity of one controller and the number of controllers resulting from Algorithm 1. Since the coverage ranges of all existing and new added controllers may overlap, the optimal solution is larger than the lower bound.

In order to measure the impact of different port allocation policies, we increase the number of ports for switch interconnection at each switch from 8 to 12. In this setting, Fig. 5(b) indicates that Algorithm 2 always outperforms the random algorithm since it redeploys fewer additional controllers to ensure the demand of 2-coverage. Additionally, the number of required additional controllers decreases, when each switch has more ports for switch interconnection, under the same datacenter size. In summary, our algorithm always utilizes fewer number of new controllers than the random algorithm to enable the demand of 2-coverage, irrespective of the datacenter size.

5.2.2 The impact of invalid controllers

This section evaluates the importance and effectiveness of the 2-coverage, where each switch is controlled by two controllers. The evaluation results show that the 2-coverage can significantly improve the fault-tolerant capability of the 1-coverage. Note that the 1-coverage problem means that each switch is controlled by one controller.

In the experiment setting, the Jellyfish datacenter utilizes 1000 switches, each of which has 24 ports. The capacity and coverage range of each controller are the same as Section 5.1.1. A given number of controllers are randomly set to



(a) A switch uses 8 ports for switches' interconnection.
(b) A switch uses 12 ports for switches' interconnection.

Fig. 7. The impact of the number of switches on the number of links in the multicast tree.

become invalid. We then observe the number of invalid switches on average, which is not controlled by any controller, under the 1-coverage and 2-coverage.

Fig. 6 reports that the amount of resulting invalid switches increases along with the increase of failed controllers under both algorithms. However, under the same number of failed controllers, the 2-coverage exhibits much fewer invalid switches than the basic 1-coverage, irrespective of the number of invalid controllers. Additionally, the similar results are achieved when each switch increases the number of ports, allocated to switch interconnection, from 8 to 12. Moreover, the number of invalid switches, due to the same number of failed controllers, increases when each switch uses more ports for switch interconnection. That is, the impact of invalid controllers is more serious during such an increasing process under both algorithms. In a summary, the 2-coverage can considerably improve the fault-tolerant capability of the 1-coverage.

5.3 Performance of the method minimizing the communication overhead of synchronization

In this section, we evaluate the cost of our method minimizing the communication overhead of synchronization among controllers, compared to prior unicast method and the steiner algorithm in [28]. For prior unicast algorithm, the sender synchronizes information to each other controller along the shortest path. In this way, the synchronization among n controllers incurs n^2 unicast transmissions. For the Steiner algorithm, a complete graph containing all multicast group members is firstly constructed, based on which a minimum spanning tree is derived. Then, each virtual link in the minimal spanning tree is replaced by its corresponding shortest path in practice. Finally, the minimum spanning tree is derived based on the resultant graph again. Specifically, unnecessary edges are deleted until all the leaves in the graph are multicast group members.

Given a datacenter, we first derive the locations of these controllers from Algorithm 1. We then construct a multicast tree to connect these controllers by Algorithm 3 and the Steiner algorithm, respectively. Thus, all controllers synchronize their states along the resultant multicast tree. To minimize the synchronization overload, it is obviously that the number of links in the multicast (Steiner) tree should be minimized.

As shown in Fig. 7, the multicast tree resulting from Algorithm 3 employs fewer number of links than prior unicast-driven synchronization method and the Steiner algorithm [28]. Thus, the synchronization among controllers incurs much less network cost via the multicast tree derived by Algorithm 3. Additionally, we find that the same results are achieved, when each switch increases the number of ports allocated for switch interconnections. Meanwhile, given the same datacenter, the resulting multicast tree utilizes fewer links, when each switch uses more ports for switch interconnection. We also note that it is not obvious that Algorithm 3 is better than the Steiner algorithm. When there are 1000 switches in the datacenter, our AMG algorithm employs 10 fewer links than prior multicast (Steiner) algorithm in Fig. 7(b). It is because that the size of datacenters is small in the experiment settings. We can see that the advantage of Algorithm 3 increases with the number of switches in a datacenter in Fig. 7. Thus, when the size of a datacenter is larger, the advantage of Algorithm 3 will be more obvious. Moreover, it is worth noting that Algorithm is faster than the Steiner algorithm in [28] to generate the multicast tree. This is important for instant applications in large datacenters.

In summary, Algorithm 3 always constructs a multicast tree for the state synchronization among controllers, with much fewer number of links than the unicast method and the Steiner algorithm. Owing to fewer links, the resultant tree can reduce the communication overhead due to the frequent state synchronization among all of controllers.

6 Related Work

For the controller's placement problem, Heller et al. firstly studied how many controllers are sufficient and where they should be deployed [25] in WAN. The propagation latency between a controller and a switch is a constraint on the number and locations of controllers. This metric alone, however, cannot characterize the real requirements of SDN applications. Yao et al. extend this work to take into consideration the load of controllers [31]. They formulate the capacitated controller placement problem that accounts for the different capacities of controllers to ensure that the controller can deal with its load at any time. However, these works all assume that each node only connects to one controller. That is to say, these placement policies do not consider the failure of controllers. Bari et al. propose two heuristics to dynamically provisioning controllers [32]. Their goals are to minimize flow setup time, control traffic and switch-to-controller reassignments. But, the dynamic assignments need more time and they also do not consider the failure of controllers.

Hock et al. also extend Heller et al.'s work [25] and propose the POCO framework to achieve the whole solution space for placements of k controllers [33]. By generating all possible placements for k controllers, the trade-offs between some metrics including inter-controller latency, load balancing between controllers, and failure resilience are explored. Recently, Lange et al. extended the POCO framework with heuristics to make it work in large or dynamic topologies [34]. However, the authors only focus on one and two failure scenarios for nodes and links and do not consider the failure of controller itself in their formulation. In addition, the number of controllers k to be deployed in the network is

given as an input parameter in the POCO framework, and that is our optimizing objective.

Hu et al. deal with controller placements from a reliability view point [35]. In this case, the authors develop several placement algorithms to make informed placement decisions and use the reliability of SDN as the placement metric. Then, the authors compare through simulation different placement algorithms [36]. To measure reliability, they propose a metric called expected percentage of control path loss. Moreover, they proof that the reliability-aware controller placement is NP-hard and analyze the trade-off between reliability and latency by testing different placement algorithms. However, similar to [33], the number k of controllers is also an input of placement algorithms.

Ros et al. have studied the southbound reliability in software-defined networks and employed connectivity as a surrogate for reliability [37].They heuristically rank facilities according to their expected contribution to southbound reliability for as most nodes as possible. Given threshold β, for every node, it has to connect to a subset of deployed controllers such that the probability of having at least an operational path is higher than the given threshold. The same authors have extended their previous work and enhanced the performance and the results of their previous work by simplifying the flow network [37]. However, the main goal of their works is to achieve high reliability in the southbound interface between controllers and nodes. In this paper, we systematically solve the minimal coverage problem, the minimal fault-tolerant coverage problem and the minimal communication overhead of synchronization among controllers, which are not considered in other works about controller placements at present. Moreover, aforementioned works mainly focus on wide-area networks or Internet topologies, but our works start from datacenters, and we hope the work in this paper to promote research on the topic.

7 Conclusion

Currently, cloud datacenters still lack a scalable and resilient control plane due to the coverage problem of controllers. In this paper, we systematically study the essential problem from three aspects. We first propose an approximation method to deploy the minimum number of controllers such that each switch is controlled by one controller. We then design a dedicated method to address the minimal fault-tolerant coverage problem. Thus, each switch will be taken over by another controller when the master one fails. Moreover, controllers have to synchronize their local network views for operating on a global network view. To reduce the resulting overhead, we minimize not only the number of involved controllers but also the cost of multicast tree used for synchronization. Extensive evaluation results show that our approaches can significantly decrease the number of required controllers, improve the fault-tolerant capability, and reduce the communication overhead of state synchronization. Moreover, these models and approaches proposed in this paper can be applied to cloud datacenters with other network topologies.

Acknowledgments

The authors thank all the anonymous reviewers for their insightful feedback. Besides, this work is partly supported by the National Natural Science Foundation for Outstanding Excellent young scholars of China under Grant No.61422214, National Basic Research Program (973 program) under Grant No.2014CB347800, the Program for New Century Excellent Talents in University, the Hunan Provincial Natural Science Fund for Distinguished Young Scholars under Grant No.2016JJ1002, the Research Funding of NUDT under Grant Nos.JQ14-05-02 and ZDYYJCYJ20140601.

References

[1] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, pp. 51–62, 2009.

[2] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "Bcube: a high performance, server-centric network architecture for modular data centers," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, pp. 63–74, 2009.

[3] D. Guo, H. Chen, Y. He, H. Jin, C. Chen, H. Chen, Z. Shu, and G. Huang, "Kcube: A novel architecture for interconnection networks," *Information Processing Letters*, vol. 110, no. 18-19, pp. 821–825, 2010.

[4] D. Guo, C. Li, J. Wu, and X. Zhou, "Dcube: A family of network structures for containerized data centers using dual-port servers," *Computer Communications*, vol. 53, pp. 13–25, 2014.

[5] D. Abts, M. R. Marty, P. M. Wells, P. Klausler, and H. Liu, "Energy proportional datacenter networks," in *Proc. 37th ACM ISCA*, Saint-Malo, France, 2010, pp. 338–347.

[6] J. H. Ahn, N. L. Binkert, A. Davis, M. McLaren, and R. S. Schreiber, "Hyperx: topology, routing, and packaging of efficient large-scale networks," in *Proc. ACM/IEEE Conference on High Performance Computing(SC)*, Portland, Oregon, USA, 2009.

[7] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: Networking data centers randomly." in *Proc. USENIX NSDI*, San Jose, CA, USA, April 2012.

[8] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, and M. Zhu, "B4: experience with a globally-deployed software defined wan," *Acm Sigcomm Computer Communication Review*, vol. 43, no. 4, pp. 3–14, 2013.

[9] C. Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven wan," *Acm Sigcomm Computer Communication Review*, vol. 43, no. 4, pp. 15–26, 2013.

[10] D. Li, Y. Shang, and C. Chen, "Software defined green data center network with exclusive routing," in *Proc. IEEE INFOCOM*, Toranto, Canada, April 2014.

[11] P. Tammana, R. Agarwal, and M. Lee, "Cherrypick: tracing packet trajectory in software-defined datacenter networks," in *ACM SIGCOMM Symposium on Software Defined NETWORKING Research*, 2015, pp. 1–7.

[12] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[13] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4, pp. 1–12, 2007.

[14] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama *et al.*, "Onix: A distributed control platform for large-scale production networks." in *Proc. USENIX OSDI*, Vancouver, BC, Canada, October 2010.

[15] O. N. Fundation, "Software-defined networking: The new norm for networks," *ONF White Paper*, 2012.

[16] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, "Towards an elastic distributed sdn controller," in *Proc. ACM HotSDN*, HongKong, August 2013.

[17] J. Xie, D. Guo, Z. Hu, T. Qu, and P. Lv, "Control plane of software defined networks: A survey," *Computer Communications*, vol. 67, pp. 1–10, 2015.

[18] A. Tootoonchian and Y. Ganjali, "Hyperflow: A distributed control plane for openflow," in *Proc. USENIX INM/WREN*, SAN JOSE,CA, April 2010.

[19] M. Jarschel, T. Zinner, T. Hoßfeld, P. Tran-Gia, and W. Kellerer, "Interfaces, attributes, and use cases: A compass for sdn," *IEEE Communications Magazine*, vol. 52, no. 6, pp. 210–217, 2014.

[20] V. Kotronis, X. Dimitropoulos, and B. Ager, "Outsourcing the routing control logic: better internet routing based on sdn principles," in *Proc. ACM HotNets*, Redmond, WA, USA, October 2012.

[21] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "Dcell: a scalable and fault-tolerant network structure for data centers," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4, pp. 75–86, 2008.

[22] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, "Logically centralized?: state distribution trade-offs in software defined networks," in *Proc. ACM HotSDN*, Helsinki, Finland, August 2012.

[23] Z. Guo, M. Su, Y. Xu, Z. Duan, L. Wang, S. Hui, and H. J. Chao, "Improving the performance of load balancing in software-defined networks through load variance-based synchronization," *Computer Networks*, vol. 68, pp. 95–109, 2014.

[24] R. M. Karp, *Reducibility among combinatorial problems*. Springer, 1972.

[25] B. Heller, R. Sherwood, and N. McKeown, "The controller placement problem," in *Proc. ACM HotSDN*, Helsinki, Finland, August 2012.

[26] M. Garey and D. Johnson, *Computers and intractability: a guide to the theory of NP-completeness*. New York: free-man, 1979.

[27] T. W. Haynes, S. Hedetniemi, and P. Slater, *Fundamentals of domination in graphs*. CRC Press, 1998.

[28] L. Kou, G. Markowsky, and L. Berman, "A fast algorithm for steiner trees," *Acta Informatica*, vol. 15, no. 2, pp. 141–145, 1981.

[29] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *Proc. ACM SIGCOMM IMC*, Chicago, Illinois, USA, August 2009.

[30] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker, "Applying nox to the datacenter." in *Proc. ACM HotNets*, New York, USA, October 2009.

[31] G. Yao, J. Bi, Y. Li, and L. Guo, "On the capacitated controller placement problem in software defined networks," *IEEE Communications Letters*, vol. 18, no. 8, pp. 1339–1342, 2014.

[32] M. F. Bari, A. R. Roy, S. R. Chowdhury, Q. Zhang, M. F. Zhani, R. Ahmed, and R. Boutaba, "Dynamic controller provisioning in software defined networks." in *CNSM*, Zurich, Switzerland, October 2013.

[33] D. Hock, M. Hartmann, S. Gebert, and M. Jarschel, "Pareto-optimal resilient controller placement in sdn-based core networks," in *Teletraffic Congress (ITC), 2013 25th International*, 2013, pp. 1–9.

[34] S. Lange, S. Gebert, T. Zinner, P. Tran-Gia, D. Hock, M. Jarschel, and M. Hoffmann, "Heuristic approaches to the controller placement problem in large scale sdn networks," *IEEE Transactions on Network & Service Management*, vol. 12, no. 1, pp. 1–1, 2015.

[35] Y. Hu, W. D. Wang, X. Y. Gong, X. R. Que, and S. D. Cheng, "On the placement of controllers in software-defined networks," *Journal of China Universities of Posts & Telecommunications*, vol. 19, no. 19, pp. 92–97, 2012.

[36] Y. Hu, W. Wang, X. Gong, X. Que, and S. Cheng, "Reliability-aware controller placement for software-defined networks," *Wireless Communication Over Zigbee for Automotive Inclination Measurement fChina Communications*, vol. 11, no. 2, pp. 672–675, 2013.

[37] F. J. Ros and P. M. Ruiz, "On reliable controller placements in software-defined networks," *Computer Communications*, vol. 77, pp. 41–51, 2016.

Junjie Xie Junjie Xie received the B.S. degree in computer science and technology from Beijing Institute of Technology, Beijing, China, in 2013. He received the M.S. degree in College of Information System and Management, National University of Defense Technology (NUDT), Changsha, China, in 2015. He is currently a PhD student in NUDT. His research interests include distributed systems, data centers, software-defined networking and interconnection networks. Email: xiejunjie06@gmail.com

Deke Guo Deke Guo received the B.S. degree in industry engineering from Beijing University of Aeronautic and Astronautic, Beijing, China, in 2001, and the Ph.D. degree in management science and engineering from National University of Defense Technology, Changsha, China, in 2008. He is a Professor with the College of Information System and Management, National University of Defense Technology, Changsha, China. His research interests include distributed systems, software-defined networking, data center networking, wireless and mobile systems, and interconnection networks. He is a member of the ACM and the IEEE.

Xiaomin Zhu Xiaomin Zhu received the BS and MS degrees in computer science from Liao ning Technical University, Liao ning, China, in 2001 and 2004, respectively, and Ph.D. degree in computer science from Fudan University, Shanghai, China, in 2009. In the same year, he won the Shanghai Excellent Graduate. He is currently an associate professor in the College of Information Systems and Management at National University of Defense Technology, Changsha, China. His research interests include scheduling and resource management in green computing, cluster computing, cloud computing, and multiple satellites. He has published more than 50 research articles in refereed journals and conference proceedings such as IEEE TC, IEEE TPDS, JPDC, JSS and so on. He is also a frequent reviewer for international research journals, e.g., IEEE TC, IEEE TNSM, IEEE TSP, JPDC, etc. He is a member of the IEEE, the IEEE Communication Society, and the ACM.

Bangbang Ren Bangbang Ren received the B.S. degree in management science and engineering from National University of Defense Technology,Changsha, China, in 2015. He is currently working toward the M.S. degree in College of Information System and Management,National University of Defense Technology, Changsha, China. His research interests include software-defined networking, data center networking.

Honghui Chen Honghui Chen received the MS degree in operational research and the PhD degree in management science and engineering from the National University of Defense Technology, Changsha, China, in 1994 and 2007, respectively. Currently, he is a professor of Information System and Management, National University of Defense Technology, Changsha, China. His research interests include information system, cloud computing and Information Retrieval.