Scalable and Reliable Orchestration for Balancing the Workload Among SDN Controllers

José Moura[1]

[1] Instituto de Telecomunicações (IT), Instituto Universitário de Lisboa (ISCTE-IUL), Lisbon, Portugal
jose.moura@iscte-iul.pt

Abstract. There is a high demand for Software Defined Networking (SDN) solutions to control emerging networking scenarios. Due to the centralized design of SDN, a single SDN controller could have its performance severely degraded due to workload congestion or even it become out of service after being menaced by failures or cyber-attacks. To enhance the robustness and scalability of the control system level, it is fundamental the deployment of multiple redundant controllers, which need to be correctly orchestrated for ensuring these controllers efficiently control the data plane. This work designs, deploys, and evaluates a new East/Westbound distributed light protocol, which supports the leaderless orchestration among controllers. This protocol is based on group communication over UDP. The evaluation results shows the merits of the proposed orchestration design on the system scalability, workload balancing and failure robustness.

Keywords: Scalable, reliability, control, leaderless orchestration, workload

1 Introduction

Emerging networking scenarios require for novel and agile solutions to control the available resources of the networked systems. SDN-based solutions are very appealing to be used for controlling the network infrastructures used in upcoming use cases. There is also a well-defined standard on the Southbound API of SDN, i.e. OpenFlow protocol. Nevertheless, further investigation is needed towards a light, scalable, and efficient East/Westbound protocol for supporting the orchestration among a group of SDN controllers, which share among them the redundant control (Ahmad & Mir, 2020) (Hoang et al., 2022). Aligned with this, the current work proposes a new multicast East/Westbound protocol that dynamically orchestrates any number of SDN controllers in a complete distributed way, and without a top-level centralized orchestrator as suggested in (Moura & Hutchison, 2022). Avoiding this top-level orchestrator, the system enhances its scalability and robustness against respectively high control workloads and any eventual failure at the single orchestrator. Using the novel solution, when a controller fails, the still available controllers apprehend that event and correctly orchestrate among them the network devices previously under the control of the failed controller. The paper structure is as follows. After the introduction, Section 2 analyses related work, highlighting the novel aspects of the current publication. Section 3 discusses the design of the proposed orchestration solution and the associated protocol. Its deployment is debated in Section 4, and Section 5 evaluates the proposed system. Section 6 discusses the main evaluation results. Finally, Section 7 concludes the paper with some future research directions.

2 Related Work

This section analyzes some related work, for highlighting the novel aspects of the current investigation. The work in (Ahmad & Mir, 2020) (Hu et al., 2018) analyzed several controller architectures considering several performance parameters: scalability, consistency, reliability, load balancing and security. The current work endorses all the previous parameters, except the security one.

The authors of (Oktian et al., 2017) state that typical services provided by East/Westbound API are the controllers' coordination towards a control leader election, the distribution of network information and mitigation of controller failovers. The current proposal for an East/Westbound protocol can be classified as a leaderless solution. It uses multicast communication for distributing the ID of each controller among the others, announcing the availability of the former controller for the remaining controllers. In this way, each controller can create the same ordered list of IDs. The order in that list of each controller can be eventually used once again by each controller orchestration function for autonomously deciding which switches will be under its control. After a controller failure, the network devices previously under the control of that failed controller are distributed again and without collisions among the available controllers.

(Hoang et al., 2022) propose a new East/West interface for sharing network state between heterogeneous SDN domains. The current work investigates the control of a single SDN domain by a set of redundant and homogeneous controllers.

(Lyu et al., 2018) developed an analytic study that stochastically optimizes at different time scales the on-demand activation of controllers, the adaptive association of controllers and switches, and the real-time processing and dispatching of flow requests. In the current deployed proposal the previous aspects are covered, except the online controller activation following the system load dynamics, which is let for future work.

3 Design

This section presents the diverse parts of our distributed solution to support the flexible and dynamic workload leaderless orchestration among any number of SDN controllers. The sub-section 3.1 discusses the orchestration protocol, and the sub-section 3.2 discusses some interesting controller orchestration functions.

3.1 Orchestration Protocol

The current sub-section presents the communication group protocol used to enable the automatic discovery of all the active controllers by each individual controller. Then, each controller can use a local orchestration function to select which switch or Packet In message that controller can exclusively control. The orchestration functions will be debated in the next sub-section. Figure 1 visualizes the communication protocol, as an example, among three controllers, but this protocol supports any number of controllers. Each controller has two light processes responsible for the transmission and reception of the orchestration East/Westbound protocol messages. The first light process of a controller sends an orchestration message to a multicast group announcing its own id, which was randomly generated at its startup. The second light process of the same controller only receives messages from that group. Each time the receiving thread of a controller receives an orchestration message, the controller verifies if the id of the announced controller is already known. When the received id is not known, then the receiving thread updates a list of discovered ids. As shown in Figure 1, this protocol requires a minimum number of multicast messages equal to the number of active controllers to enable the full awareness in each controller about all the other controllers.

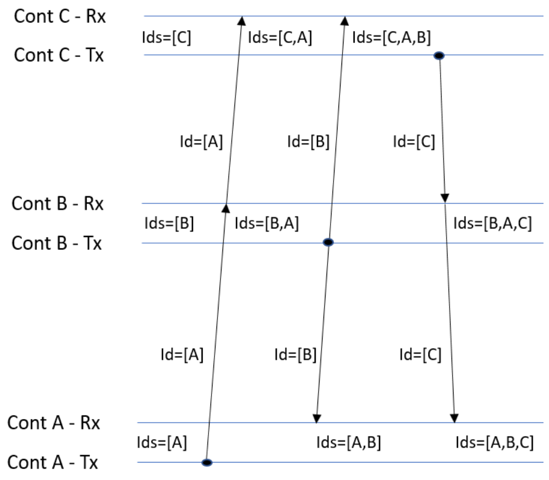


Figure 1. Proposed Orchestration Protocol

3.2 Controller Orchestration Function

After the receiving thread of a specific controller has collected the ids of all the active controllers, this thread stores the collected ids in a list, which is shared with the main process of the controller. Then, the controller processes this list of ids obtaining the total number of active controllers (i.e. the list length) and the individual order of the current controller in that list. These two parameters are fundamental to the evaluation of the controller orchestration function visualized in Expression 1, where *dpid* is the unique datapath id associated to each switch.

$$\text{dpid mod(num\_server) == order} \qquad (1)$$

When the equality in (1) becomes True, this occurs exclusively at a single controller among any set of controllers. Therefore, there is always a unique controller to decide how the message within the Packet-In received from switch *dpid* should be analyzed and processed. In this way, there is a distributed decision or consensus mechanism among the controllers. This solution offers the significant advantage of avoiding the exchange of OpenFlow or other extra synchronization messages among controllers to reach the final decision in which controller should perform the control decision.

The previous orchestration function in use cases where there are distinct amounts of data flows traversing the forwarding switches could not be totally fair in terms of balancing the load among the controllers. To mitigate this problem, it is now discussed an alternative function, which could be fairer than the previous discussed function. This alternative orchestration function for each controller could decide if it processes or not any received Packet-In message is summarized in (2). The subtle difference in relation to (1) is the replace of *dpid* by *packet_in_counter*, which is the aggregated value of all received Packet-In messages by each controller. Assuming every switch is simultaneously connected via OpenFlow with every available controller, all the controllers share the same trend on the *packet_in_counter* parameter. The decision algorithm in (2) enables a fairer control load distribution among the diverse controllers, but it has a potential drawback. It can increase the number of times each switch must change from controller. Each time a new controller assumes the control of a switch, the controller must delete the old rules and install new ones on that switch. All this OpenFlow traffic increases the overload on the control channel. Alternatively, using the first function, in (1), each switch is always controlled by the same controller during all the time. In this way, the channel control will not become so congested as in the case of function (2). There is here clearly a tradeoff between the orchestration fairness and control channel overload.

$$\text{packet\_in\_counter mod(num\_server) == order} \qquad (2)$$

4 Deployment

The current Section details in the next sub-sections the deployment of the orchestration protocol and controller orchestration function, which were presented in Section 3.

4.1 Orchestration Protocol

As already explained in sub-section 3.1, each controller has two threads responsible for managing the orchestration protocol. As shown in Algorithm 1, inside the constructor of the controller code, these two threads are started in steps 7 and 9. The thread t1 is responsible for the periodic transmission of orchestration messages, announcing the controller id, which is randomly generated in step 4. The second thread, t2, is responsible for listening the multicast orchestration messages sent by other controllers. The steps 19-23 are where the thread t1 tries every 2 s to send a multicast message announcing its ID. Then, each controller has a listening thread (t2), steps 26-41, which receives all the multicast messages sent by the other controllers. It also decides, at a convenient time, when the list of IDs needs to be updated and announced to the local controller. The controller every second invokes the function in steps 46-48 to get a fresh version of the list containing the IDs of all active controllers. When a controller sees a list with only its own ID, it concludes that is alone and changes its role from EQUAL to MASTER. Otherwise, the controller using a local coordination function (see Algorithm 2 below) can select the network devices or data plane messages, which can be exclusively controlled by that controller.

Algorithm 1: Each controller initially starts two threads which manage the multicast communication with remaining controllers.

```
1:   from transmitter_multicast import Controller_Multicast
2:   import threading
3:   def __init__(self, *args, **kwargs):
4:      self.cont_id = str(random.randint(0, 1000))
5:      self.tx_mult = Controller_Multicast(self.cont_id)
6:      self.t2 = threading.Thread(target=self.tx_mult.receive, args=(self.tx_mult.get_id(),))
7:      self.t2.start()
8:      self.t1 = threading.Thread(target=self.tx_mult.send, args=(self.tx_mult.get_id(),))
9:      self.t1.start()
10:  end function
11:  Class Controller_Multicast(object):
12:     def __init__(self, id):
13:        self.MY_ID = id
14:        self.list_ids = []
15:     end function
16:     def send(self, id):
17:        multicast_addr = '224.0.0.1'
18:        port = 3000
19:        while True do
20:           sock.sendto(json.dumps([id]).encode('utf-8'), (multicast_addr, port))
21:           time.sleep(2)
22:        end while
23:     end function
24:     def receive(self, id):
25:        while True do
26:           cnt = cnt + 1
27:           data, address = sock1.recvfrom(256)
28:           l_rx = json.loads(data.decode('utf-8'))
29:           for i in range(len(l_rx)) do
30:              if l_rx[i] not in l_tmp:
31:                 l_tmp.append(l_rx[i])
32:              end if
33:           end for
34:           if cnt > (len(l_tmp) + 2)
35:              l = l_tmp
36:              l_tmp = [id]
37:              cnt = 0
38:           end if
39:           l.sort(reverse=False)
40:           self.list_ids = l
41:        end while
42:     end function
43:     def get_id(self):
44:        return self.MY_ID
```

```
    end function
    def get_list_ids(self):
        return self.list_ids
    end function
```

4.2 Controller Orchestration Function

Algorithm 2 shows the network function running in each controller, which enables a distributed coordination among all the SDN controllers operating in the role EQUAL. This algorithm avoids potential conflicts among the controllers (steps 5-6). Diverse orchestration functions were discussed in sub-section 3.2.

Algorithm 2: The controller assumes the role EQUAL avoiding conflicts with other controllers

```
1:  for each Packet-In Event with pkt do
2:      datapath = Event.msg.datapath
3:      dpid = datapath.id
4:      if self.mode == 'EQUAL':
5:          if not (dpid % int(self.num_serv) == int(self.order)):
6:              return
7:          else:
8:              Analyses, processes and controls the current message
9:          end if
10:     end if
11: end for
```

5 Experimental Results

This section presents the results of all the experiments conducted in this study. The results are offered based on three sets of experiences, which are listed in Table 1, and using the experimental setup visualized in Figure 2 and further detailed in Table 2. The system under evaluation is formed by m redundant controllers, n switches, n hosts and 2n-1 data plane links. The data plane was emulated by Mininet. For all the tests of the current paper, ten controllers, ten switches and nineteen data plane links were used. The controller logic, using the Ryu Library, including the auxiliary class which deploys the behavior of the orchestration communication threads, was implemented in Python3. The next sub-sections discuss the diverse obtained results.

Table 1. Evaluation tests and their main goals

Subsection	Main aspect(s) under analysis
5.1	It verifies the correct behavior of the new proposed protocol to orchestrate any number of controllers in a completely distributed way
5.2	It compares two possible controller orchestration functions for the distributed orchestration design; it also analyzes their impact on the system performance and the fairness in how each function balances the control workload among available redundant controllers
5.3	It compares two distinct designs (centralized vs. distributed) for the orchestration part of the system; It can help to reach some conclusions in terms of how each solution enables both system scalability and system resiliency

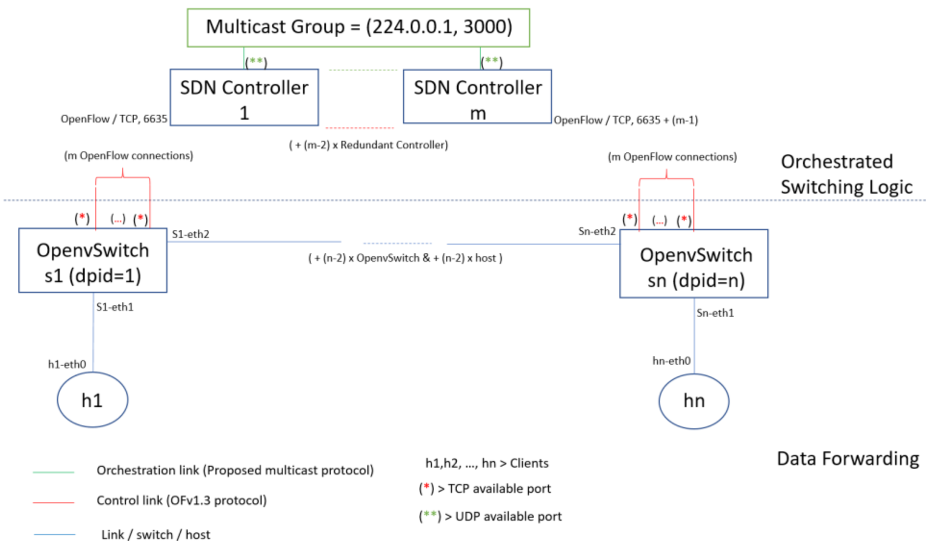


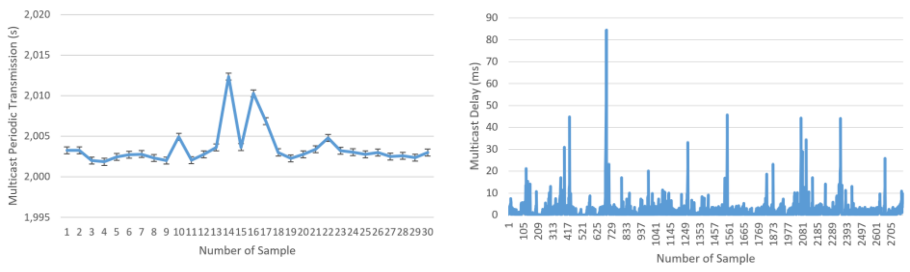
Figure 2. The SDN-based system under evaluation.

Table 2. Hardware and software tools used during the evaluation tests

ASUS Intel® Core™ i7-3517U CPU @ 1.90GHz 2.40GHz, 12 GB RAM, Windows10 Education x64	-
VirtualBox Ubuntu 22.04	https://www.virtualbox.org/; https://releases.ubuntu.com/22.04/
Ryu SDN Controller (v4.34)	https://ryu-sdn.org/
OpenvSwitch (v2.16.90, DB Schema 8.3.0)	https://www.openvswitch.org/
Python 3.9.12	https://www.python.org/downloads/release/python-3912/
Mininet (v2.3.0)	https://github.com/mininet/mininet
Wireshark (v3.4.9)	https://www.wireshark.org/

5.1 Functional Test of the Orchestration Protocol

This functional test was made using the experimental setup presented in the beginning of the current Section. The goal of this initial test was to verify if the orchestration protocol messages were periodically sent from the same controller, announcing its ID to the other controllers. Figure 3.a) visualizes some samples of the time interval between two consecutive transmissions from the same transmission thread. The time interval is roughly around 2 s. In addition, it was measured the multicast transmission delay between each multicast transmission thread and each receiver. The obtained samples are visualized in Figure 3.b). Except some sporadic peak delays within the range of [15,85]ms, the most significant amount of delay samples are below 10ms. In this way, the orchestration protocol was successfully verified.



a) Multicast Periodic Transmission   b) Multicast Transmission Delay

Figure 3. Diverse Characteristics of the Multicast Orchestration Protocol

5.2 Functional Test Comparing Two Alternative Orchestration Functions

In this experiment, the two alternative orchestration functions discussed in sub-section 3.2 were compared, using the same network traffic load, in terms of the total number of Packet In (PI) messages and the processed PI messages by each controller of the ten controllers under this test. The orchestration protocol among controllers was always the one multicast-based discussed in sub-sections 3.1 and 4.1. The obtained results are summarized in Table 3. Analyzing and comparing the results relative to the two orchestration functions, the first conclusion is that the ID orchestration function when compared with the PI orchestration one reduces significantly (from 306 to 198) the total number of PI messages during the test. Nevertheless, the former function has a lower Jain Fairness Index, i.e. 0.929, against the impressive 0.999 of the latter function. In this way, by running the current test, the tradeoff discussed in sub-section 3.2, when the two orchestration functions were initially analyzed, it was experimentally confirmed.

Table 3. Considering the two orchestration functions under comparison, PI total and processed messages are listed for each controller

	PI Orchestration Function (Jain Fairnes Index=0.999)		ID Orchestration Function (Jain Fairnes Index=0.929)	
Controller	**PI_Total**	**PI_Processed**	**ID_Total**	**ID_Processed**
1	306	31	198	15
2	306	30	198	21
3	306	31	198	27
4	306	31	198	23
5	306	30	198	25
6	306	31	198	19
7	306	30	198	17
8	306	31	198	11
9	306	31	198	13
10	306	30	198	27

5.3 Functional Test Comparing Centralized vs. Distributed Orchestration

After the fairest orchestration function among controllers was found (i.e. PI, sub-section 5.1), this function was applied to two distinct orchestration designs. The two designs under comparison are the distributed orchestration design that was discussed in sub-sections 3.1 and 4.1, and the centralized orchestration design investigated in (Moura & Hutchison, 2022). The comparison results are visualized in Figure 4. Analyzing these results, the distributed design based on multicast communication among the controllers has a lower load rate (1.6Kb/s) than the centralized design based on TCP communication (18Kb/s). Consequently, the distributed design is more scalable than the centralized one. In addition, the distributed design does not have the issue of the single point of failure that could easily occur in the centralized design.

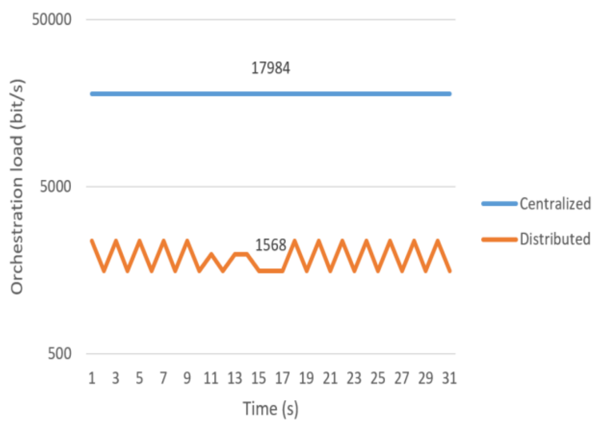


Figure 4. Centralized vs. Distributed Orchestration Design (vertical axis in logarithmic scale)

6 Discussion

This section debates the main conclusions and lessons learned during the current investigation work. From the obtained results, it was initially demonstrated that it is possible the usage of a multicast communication protocol to coordinate a set of redundant controllers for balancing in a correct way the control workload among them. This proposed distributed design for the system orchestration part clearly offers system operational advantages in comparison with an existing centralized alternative of the literature (Moura & Hutchison, 2022). The advantages provided by the new proposed leaderless orchestration design are high system salability (e.g. for both number of controllers and number of controlled network devices) and increasing robustness against system threats, such as faults or cyber-attacks. The higher scalability of the distributed design in relation to the centralized option is visible from Figure 4, with a significant reduction (from 18Kb/s to 1.6Kb/s) on the system overload induced by controllers' orchestration.

The current paper has also compared two possible controller orchestration functions for the distributed orchestration design. Both orchestration functions have been studied in terms of their impact on the system performance and the fairness in how each function balances the control workload among the available redundant controllers. Each function has a strong and a weak performance aspect, which are reversed, considering the other alternative function under study. As an example, if the owner of a network infrastructure is concerned with the amount of system resources used by the extra control / orchestration messages, trying to diminish the system energy consumption, our work indicates that the more suitable orchestration function is the one which associates to each switch a dedicated controller, considering the unique datapath id of that switch.

In the case of a controller failure, the network devices previously under the control of that failed controller are distributed again and without any collision among the still available controllers. The system could require some seconds (i.e. around 3 or 4s) to reach a coherent state among all the controllers, but after that instant, the system operates without any issue.

7 Conclusion

In this study, it is proposed a new East/Westbound leaderless protocol among the controllers to coordinate among them the control workload of a common network infrastructure being redundantly controlled by those controllers. As shown in the result section, this proposal is viable, scalable, and it offers increasing resilience against system threats due to the distributed characteristic of that proposal. For future work, the on-demand (de)activation of controllers will be investigated (Lyu et al., 2018).

References

Ahmad, S., & Mir, A. H. (2020). Scalability, Consistency, Reliability and Security in SDN Controllers: A Survey of Diverse SDN Controllers. *Journal of Network and Systems Management*, *29*(1), 9. https://doi.org/10.1007/s10922-020-09575-4

Hoang, N.-T., Nguyen, H.-N., Tran, H.-A., & Souihi, S. (2022). *A Novel Adaptive East-West Interface for a Heterogeneous and Distributed SDN Network*. https://doi.org/10.3390/electronics11070975

Hu, T., Guo, Z., Yi, P., Baker, T., & Lan, J. (2018). Multi-controller Based Software-Defined Networking: A Survey. *IEEE Access*, *6*, 15980–15996. https://doi.org/10.1109/ACCESS.2018.2814738

Lyu, X., Ren, C., Ni, W., Tian, H., Liu, R. P., & Guo, Y. J. (2018). Multi-Timescale Decentralized Online Orchestration of Software-Defined Networks. *IEEE Journal on Selected Areas in Communications*, *36*(12), 2716–2730. https://doi.org/10.1109/JSAC.2018.2871310

Moura, J., & Hutchison, D. (2022). Resilience Enhancement at Edge Cloud Systems. *IEEE Access*, *10*, 45190–45206. https://doi.org/10.1109/ACCESS.2022.3165744

Oktian, Y. E., Lee, S. G., Lee, H. J., & Lam, J. H. (2017). Distributed SDN controller system: A survey on design choice. *Computer Networks*, *121*, 100–111. https://doi.org/10.1016/J.COMNET.2017.04.038