FRACTAL: A Framework for Recursive Abstraction of SDN Control-Plane for Large-Scale Production Networks

Myungchul Kwak, Junho Suh, Ted "Taekyoung" Kwon
School of Computer Science and Engineering
Seoul National University, Seoul, Korea
Email: {mckwak, jhsuh, tk}@mmlab.snu.ac.kr

***Abstract*—Software-defined networking (SDN) renovates traditional networking systems by replacing a distributed, per-switch control plane with a (logically) centralized one. To design a scalable, highly available SDN control plane, it is inevitable to disseminate the network state to multiple instances horizontally by using measures like replication and partitioning. However, some recent studies reported that it is not sufficient to cover a large scale network in a purely horizontal manner. In this paper, we propose FRACTAL, a framework for recursive abstraction of SDN control-plane, to address this problem. In FRACTAL, a large network is divided into multiple small networks, each of which is abstracted as a single virtual switch. This "divide-and-abstract" process is recursively iterated until a divided network can be handled by a single controller. A virtual switch is controlled by the higher level controller over OpenFlow, so that FRACTAL can coexist with other SDN mechanisms.**

Keywords—Distributed Architecture, Software Defined Networking (SDN)

I. INTRODUCTION

Software Defined Networking (SDN) decouples the control planes from data planes of switches; it replaces a distributed, per-switch control plane with a (logically) centralized one on which SDN applications can control an operational network with a global network-wide view by enforcing packet forwarding rules to the distributed data planes.

From an architecture perspective, the scalability, resilience, and responsiveness is required by designing a (logically) centralized SDN control-plane to cover a large sized network for future carrier-grade cloud or cloud 2.0 services. In such large scale networking environments, it is inevitable that a (logically) centralized SDN control plane should be distributed to multiple controllers, each of which is in charge of its own partition. Many research studies have been done to address the scalability issue [4, 8], most of which leverage *replicating* and *partitioning* a network state in the control plane horizontally.

However, some of recent investigations pointed out that the horizontal (or flat) distribution of network state is not so effective for large-scale production networks, e.g., inter-datacenter networks, multi-site enterprise networks, and wide area networks, due to the latency taken by querying or replicating the network state among controllers. [9] focused on the SDN controller placement problem – how many controllers are needed and where they should be located to cover the whole network. Further, [10] illustrated how the level of consistency impacts the performance and complexity of network control logics. In case of low complexity, the inconsistent network state information is likely to lead to suboptimal decisions. If any tolerance against inconsistency is to be allowed, highly complex control is required.

978-1-4799-7899-1/15/$31.00 ©2015 IEEE

Meanwhile, [12] took a novel approach for scalability, where the SDN control plane is hierarchically organized and hence the locality is exploited for local events. It effectively limits the propagation scope of events at the control plane, reducing the amount of the network state information transmitted or replicated. However, it requires (i) the redesign of the existing SDN control plane, e.g., services and network control logics, and (ii) the development of a new protocol between parent-child SDN controllers. To the best of our knowledge, there is still no solution to take into consideration both the locality and the distributed control plane at the same time.

Inspired by the literature, in this paper, we present FRACTAL: a **F**ramework for **R**ecursive **A**bstraction of SDN **C**on**T**rol plane for large-sc**AL**e production networks. FRACTAL leverages *recursion* to achieve scalability. The key idea behind FRACTAL is that a large network is partitioned into multiple small networks; a small network can be further divided into smaller networks. This process can be recursively repeated depending on the whole network size, and the capacity of a controller. A partitioned network at any hierarchical level is called a *domain network*, which is controlled by a *domain controller*. Thus, the central idea of FRACTAL is to abstract a domain network to its parent controller as a single big virtual switch over OpenFlow.

The gain of FRACTAL is threefold. First, since FRACTAL partitions a single large network into multiple small networks, and utilizes a locality by organizing domain controllers hierarchically, it effectively eliminates the overhead of disseminating or replicating the information of events that are locally significant. Second, FRACTAL transparently abstracts a domain network as a single virtual switch that establishes a connection to its own SDN controller through the same southbound protocol like OpenFlow. Hence, there is no need to develop a new protocol between the controller and its switches. Third, FRACTAL no longer introduces the modification to existing services such as a host tracker, a topology manager, a switch manager, and other SDN applications due to the properties of FRACTAL, e.g., transparency.

II. BACKGROUND

Before elaborating on FRACTAL, We investigate a distributed SDN control plane to answer such a question–*how does a horizontally distributed SDN control plane impact the performance of SDN applications.*–which is a main motivation of this paper.

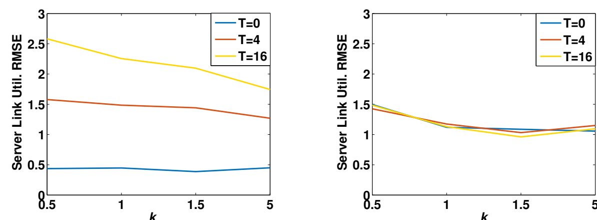
Simulation: To answer this question, we first carry out simulation experiments by leveraging a custom flow-level simulator with identical settings to [10], including the topology (comprised of two switches, and two clients and servers connected to each switch), the sync overhead, and the controllers. When a flow request (to any of two servers) arrives at switch i (i is 1 or 2), the corresponding controller i decides to which server the flow is set up–the objective is to minimize the maximum link utilization in our network. If the controller has a global network-wide view by combining both the physical network state from within its domain as well as the link utilization of the other domain and it chooses the path with the lowest maximum link utilization, it is a simple link balancing controller (LBC). Note that it is likely that the global network-wide view is potentially stale. Whereas, if a controller is aware of the (potentially stale) global network-wide view and it has a logic to tolerate [10], this is a separate state link balancing controller (SSLBC).

We vary the workload (i.e., arrivals of flow requests) using exponentially distributed flow inter-arrival times (average is 10 unit time) and Weibull distributed flow durations. We also vary the sync overhead ($T = \{0, 4, 16\}$ in simulation unit time); $T = 0$ means the network state is instantly shared between the controllers. To change the distribution of flow durations, the shape parameter is varied ($k = \{0.5, 1, 1.5, 5\}$) with the fixed scale parameter ($\lambda = 10$ in simulation unit time). Note that λ is the average flow duration. Thus, as k decreases, the frequency of flows whose duration is less than the average increases. We measure a root mean squared error (RMSE) of the utilization of the links to the servers to compare the performance of LBC and SSLBC. For the details of the simulation experiments, refer to [10].

Figure 1(a) shows that the server link RMSE is increased significantly as we increase the sync interval of a distributed SDN control plane. This means that LBC is limited since it receives the state of the links (of the other controller) after T unit time, which may be stale. Consequently, SDN applications requires a logic to tolerate the staleness, resulting in increasing the complexity of SDN applications' logic. Figure 1(b) shows that SSLBC (which seeks to maximize the local link utilization) is much more robust against staleness than LBC due to its logic to tolerate.

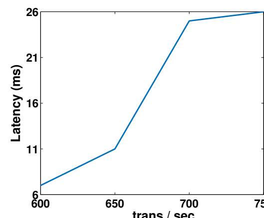
Moreover, Figure 1(a) also shows that as k increases, the performance of LBC and SSLBC are more robust against the staleness of a distributed SDN control plane. This implies that we should utilize the locality of events for latency-sensitive SDN applications by limiting the propagation scope of short flows.

Microbenchmark: Since the simulation settings are not so realistic, we further carry out experiments on a real testbed. To this end, we build a cluster of three physical servers, each of which is running OpenDaylight. Note that OpenDaylight leverages Infinispan [3], which is a distributed in-memory key/value data store to implement a distributed SDN control plane. When a controller receives the network event, the updated state is replicated to the other controllers through Infinispan.

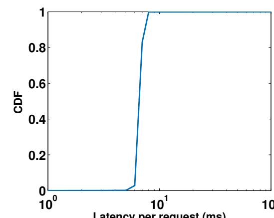


(a) The imbalance between the two server links is worsened as the sync interval T increases.

(b) SSLBC is not affected by the sync interval since its logic is independent of the staleness of network state.



(c) As the number of transactions per second increases, the synchronization (sync) delay increases due to computational overhead.



(d) CDF of the computation latencies of transactions is plotted when the transactions per second is 750.

Fig. 1: Upper figures are simulation results to demonstrate the performance issues in the horizontally distributed controllers. Lower figures are microbenchmark results on a real testbed.

To stress this data store, we use Cbench [1] on a test machine, which emulates a bunch of switches and sends `packet_in` messages, and receives `flow_mod` messages (from controller 1) to modify the flowtable. The workload to the Infinispan can be determined by varying the number of switches emulated and the rate of new flows generated at each switch. We vary the number of transactions per second (trans/sec= $\{600, 650, 700, 750\}$) to the data store.

In Figure 1(c), the sync delay (from the moment of a `packet_in` arrival at Infinispan to the moment of finish of synchronization among the controllers) increases as we increase transactions per second. At 750 transactions per second, the sync delay to synchronize the distributed SDN control plane is approximately 25 ms. As the average flow duration in production data center networks is around 100 ms [7], it is 25% overhead. However, Figure 1(d) is the cdf of the computation latencies (which is sync delay - queueing delay) at 750 transactions per second, showing that almost 90% of transactions take a few milliseconds of computation. This indicates 25 ms overall delay in synchronization is significant compared with the latency due to computation only.

III. FRACTAL DESIGN

FRACTAL aims at building a scalable framework for an SDN control plane by partitioning a large-scale network into multiple small networks recursively. A partitioned network

is abstracted as a *virtual* switch, which is controlled by a single controller. Note that the higher level controller (of the whole network) controls small networks as virtual switches through the same southbound protocol (e.g., OpenFlow). Figure 2 illustrates how a network is hierarchically abstracted by FRACTAL. We now describe the details of a "Domain Manager" which is a main component of FRACTAL.

A. Domain Manager

To transparently abstract a domain network as a *virtual switch*, we exploit a control plane of software switches (e.g., [2, 5]) so that a domain controller of a domain network can serve as an OpenFlow switch. The domain manager realizes a *virtual switch*, and consists of a connection manager, a state manager, and a configuration manager.

The connection manager manages OpenFlow connection instances with other controllers, checks the status of the connections by keep-alive messages, and processes OpenFlow messages. The state manager maintains the internal state (of a *virtual switch*) such as flow tables in which rules to be enforced for the domain network, ports from which virtual ports are built, and statistics of flows and virtual ports of its domain network. Moreover, the domain manager provides a functionality for querying and configuring the *virtual switch* through the configuration manager. That is, the configuration manager provides the interfaces for adding (deleting) virtual ports to the virtual switch, installing (retrieving) rules, and so on.

B. "Many-to-One" Mapping

When a domain controller provides transparent abstraction (for its next higher level controller), a "many-to-one" *mapping* between the switches in its domain network and a virtual switch is required; in particular, there are two types of translations: (i) topology translation and (ii) message translation.

Topology Translation: The first step for the topology translation is to specify the relations between the switches of a domain network and a *virtual switch*. For this, a domain manager maintains a bidirectional hash map consisting of a pair of `(vsw-id, vport#)` and `(sw-id, port#)` which is specified by a configuration manager in the above. `(sw-id, port#)` means the actual physical port which is exposed to higher level controller, and `(vsw-id, vport#)` means the corresponding virtual information for the port. Further, the map provides a straightforward translation for the topology.

Message Translation: Since a domain network is seen as a *virtual switch* to other (domain) controller, all non-local messages triggered on a switch in the domain network should look like the messages triggered on the *virtual switch*. In OpenFlow, two types of messages are believed to be influenced by this message translation: *switch-to-controller* messages such as `packet_in` and *controller-to-switch* messages such as `packet_out` and `flow_mod`.

For example of "*many-to-one*", assume that there is a domain network comprised of a switch with two ports, and the mapping table in its domain controller has next two entries: [$(switch_1, port_1)$↔$(vswitch_1, port_1)$] and [$(switch_2, port_2)$↔$(vswitch_1, port_2)$]. When a `packet_in` message is triggered from $port_1$ of $switch_1$, it is to be exported to the higher level domain controller, the message translator in the domain manager rewrites the `in_port` field in the `packet_in` message with $vport_1$ of the $vswitch_1$ by looking up the mapping table. In the opposite direction, it rewrites the `out_port` field in the `packet_out` message with $port_1$ of $switch_1$. However, the message translation of `flow_mod` message is not straightforward, since it is to be interpreted in the *virtual switch* rather than a domain network. For this, the message translator interacts with local SDN applications such as the topology manager to transform the rules written for the *virtual switch* into ones for the domain network.

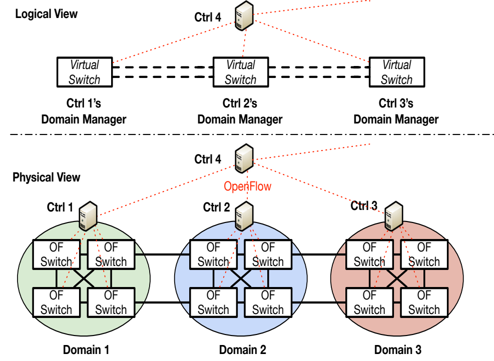


Fig. 2: A large-scale network is partitioned into three domain networks, which form a two-level hierarchy in FRACTAL. Red dotted lines indicate Openflow connections.

Moreover, in FRACTAL, only the links between the virtual ports have to be exposed to the higher level controller. Thus, the controller of a lower level domain network has to redirect the LLDP (Link Layer Discovery Protocol) packets incoming at virtual ports. The translator takes over this role, it virtualizes and redirects these LLDP packets to its higher level controller by the above process. We call this mechanism *virtualized* LLDP dissemination.

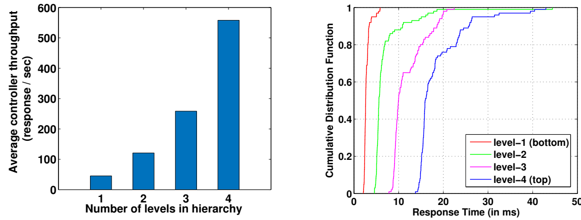
IV. Evaluation

In this section, first we explain a microbenchmark used to evaluate the scalability and the performance overhead of FRACTAL. We then describe how FRACTAL performs on a real network considering an SDN application for traffic engineering on data center network.

A. Microbenchmark

Scalability: To evaluate the scalability, we use Cbench [1] which stresses FRACTAL by varying the number of switches that send a number of requests, and measures the per-switch responsiveness as we vary the number of controllers that comprise FRACTAL.

Figure 3(a) shows the controller throughput as the level of the controller hierarchy increases (from 1 to 4). The controller throughput means the number of `packet_in` messages processed by a bottom level controller divided by the number of switches that the controller manages. Assuming 80 switches in the whole network, we partition the network by adopting



(a) Average (per-switch) controller throughput is plotted as the level of controller hierarchy varies.

(b) CDF of response times between switches and controllers is plotted when the hierarchy level is 4.

Fig. 3: Microbenchmark result to evaluate the scalability and the performance overhead of FRACTAL.

binary tree as a hierarchical structure. For instance, if the hierarchy level is 1, a single controller handles all the 80 switches. If the level is 4, each bottom level domain controller handles only 10 switches (there are 8 bottom level controllers). The per-switch controller throughput superlinearly increases as the hierarchy level grows since the FRACTAL framework divides the network, in which a single controller can in turn process relatively much more messages in the per-switch perspective. Note that the controller throughput of inter-domain flow is originally distributed among the higher level controller, but we omit it since it takes a small portion compared to intra-domain ones.

Performance Overhead: Partitioning a network incurs some overhead to the FRACTAL framework. In abstracting a domain network, its domain manager looks up the mapping table for the topology and rewrites the messages for the context of the *virtual switch*. Notice that there is no additional overhead to the data plane; packets are forwarded at line rate. FRACTAL adds no additional traffic in the control plane either. FRACTAL only adds the propagation delay between the hierarchy levels and the processing overhead to the messages that requires mapping and resolution. Note that the propagation delays can be omitted, because they are added only for inter-domain flows that take a small portion.

To quantify the overhead due to partitioning, we measure the response time between `packet_in` and the corresponding `flow_mod` when the level of controller hierarchy is four. Figure 3(b) evaluates the performance overhead that partitioning incurs. The response time increases by 5 ms on average for crossing each level in the hierarchy. Thus, there is a tradeoff between the per-switch controller throughput and the response time of a non-local message in determining the level of controller hierarchy.

B. *Experiments on campus network*

A data center topology: We now build a data center network on our campus network; two island networks are connected on our campus network as an overlay. That is, one network contains a ($k = 4$, where k is the number of pods per core switch and a pod consists of two layers of $k/2$ switches) FatTree topology connected with the other through tunnels, both of which forms a $k = 8$ FatTree topology with 16 tunneling links between the networks. There are three domain controllers: one for each network for local events and the 3rd one on the campus network for non local events.

For the purpose of comparison, we consider another network setting, which is a single large *non-blocking* switch. This setting is configured to find out the maximum throughput since traffic is constrained only by link speeds. Because non-oversubscribed FatTree topologies are rearrangeably non-blocking [6], a non-blocking FatTree topology can also achieve the optimum performance.

A traffic engineering on data center topology: A traffic engineering (TE) can highlight the SDN's benefit because the controller has a global network state such as a topology, flow statistics and link utilization. The controller can thus arbitrarily choose best paths even if they are not shortest, with no concerns about convergence time, forwarding loops, or black holes.

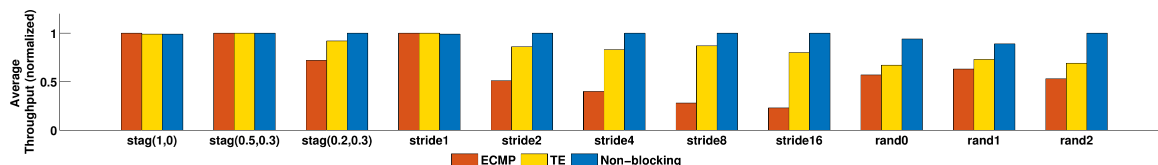
In TE (or SDN in general), fast detection and scheduling are crucial to the efficient network utilization. The fast detection of small size flows is well studied in [11]. Further, fast scheduling of flows on large-scale networks is well studied in Hedera [6]. In this paper, we use a polling mechanism to gather the network state and run TE every 1 second. For implementation simplicity, we use the *global first fit* algorithm [6] for rerouting.

By default, all traffic in our network follows shortest paths—we do not use Spanning Tree Protocol or equivalent. When multiple shortest paths exist, ties are broken by using equal-cost multi-path (ECMP) hashing based on the TCP/IP 5-tuple. At every second, TE gathers the utilization of every link in the network and detects congested links and the corresponding flows. TE then reschedules all flows routed to the congested links to less utilized ones.

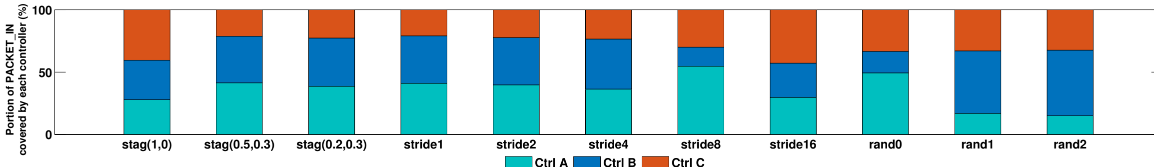
We measure the aggregate data throughput as we vary spatial workload patterns with the TE mechanism above. In all cases, we employ the same TE algorithm (Hedera). Meanwhile, we measure the total number of bytes sent (data traffic) and the fraction of those bytes that we can schedule on alternate routes in each case.

Workloads: We use a workload generator originally written for Hedera [6]. It has three different traffic patterns such as *Stride(s)*, *Staggered Prob (EdgeP, PodP)*, and *Random(u)*. We exploit the same traffic patterns with [6]. For our performance baseline we use shortest-path ECMP forwarding with no TE. Flow sizes are exponentially-distributed with 1 GB average. Flow inter-arrival times are exponentially distributed with 1 ms average.

Results: Figure 4(a) shows the (data traffic) throughput of various flow patterns on the our data center topology and on the ideal single non-blocking switch. Note that even the non-blocking switch does not achieve maximum throughput (1.0) since sometimes two hosts transmit packets to the same destination, causing congestions. Although a FatTree topology is rearrangeably non-blocking, there is a significant gap between ECMP forwarding and the single non-blocking switch due to collisions where multiple flows are (hashed and) forwarded onto the same link. This degradation is mitigated by TE due to its flow scheduling.



(a) Normalized aggregate throughput on the data center topology in *FatTree* shows that TE outperforms ECMP for various traffic patterns.



(b) The number of `packet_in` messages handled by each controller: $Ctrl\ A$ and $Ctrl\ B$ mean the lower level controllers which take on each k=4 *FatTree* network, $Ctrl\ C$ means the higher level controller which takes on the partitioned domains.

Fig. 4: For various traffic patterns, aggregate throughput for ECMP and TE on the k=8 FatTree topology (with FRACTAL) is compared to a single non-blocking switch.

We also counts the number of `packet_in` messages sent to each domain controller in each domain network, to see whether the overall control overhead is efficiently distributed among controllers. Figure 4(b) shows the number of `packet_in` messages processed by each controller with the same scenario for Figure 4. $Ctrl\ A$ and $Ctrl\ B$ mean the results of the lower level controllers, and $Ctrl\ C$ means the result of the higher level controller. Figure 4(b) demonstrates that the three controllers split the entire control overhead moderately. In the stag(1,0) scenario, there should be no `packet_in` messages to $Ctrl\ C$ if we count only the `packet_in` messages for data flows.

V. Conclusions

In this paper, we present FRACTAL, a framework for scalable dissemination of the network state over the control plane. The key idea of FRACTAL is to "divide-and-abstract" a large network recursively until a divided network can be fully handled by a controller. FRACTAL presents a tradeoff between the message processing delay over the controller hierarchy and the control plane throughput. We demonstrate the benefit of FRACTAL by building a testbed emulating a data center with the open source SDN controller, OpenDaylight.

Acknowledgement

This research was supported by Basic Science Research Program through the "National Research Foundation of Korea(NRF)" funded by the Ministry of Science, ICT & future Planning (2013R1A2A2A01016562). We appreciate the inspiring idea of grouping switches from Dr. Younghwa Kim and Dr. Saehyoung Park with ETRI.

References

[1] Cbench. http://archive.openflow.org/wk/index.php/Oflops.
[2] Indigo project. http://www.projectfloodlight.org/indigo/.
[3] Infinispan: distributed in-memory key/value data grid and cache. http://infinispan.org.
[4] Open daylight project. http://www.opendaylight.org/.
[5] Open virtual switch. http://http://openvswitch.org.
[6] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 19–19, Berkeley, CA, USA, 2010. USENIX Association.
[7] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.
[8] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar. Onos: Towards an open, distributed sdn os. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 1–6, New York, NY, USA, 2014. ACM.
[9] B. Heller, R. Sherwood, and N. McKeown. The controller placement problem. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 7–12, New York, NY, USA, 2012. ACM.
[10] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann. Logically centralized?: State distribution trade-offs in software defined networks. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 1–6, New York, NY, USA, 2012. ACM.
[11] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca. Planck: Millisecond-scale monitoring and control for commodity networks. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 407–418, New York, NY, USA, 2014. ACM.
[12] S. Schmid and J. Suomela. Exploiting locality in distributed sdn control. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 121–126, New York, NY, USA, 2013. ACM.