# Toward Highly Available and Scalable Software Defined Networks for Service Providers

Dongeun Suh, Seokwon Jang, Sol Han, Sangheon Pack, Myung-Sup Kim, Taehong Kim, and Chang-Gyu Lim

The authors review the state of the art on high availability and scalability issues in SDN and investigate relevant open source activities. In particular, two well-known open source projects, OpenDaylight (ODL) and Open Network Operating System (ONOS), are analyzed in terms of high availability and scalability. They also present experimental results on the flow rule installation/read throughput and the failover time upon a controller failure in ONOS and ODL, and identify open research challenges.

This work was supported by the Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (B0101-15-233: Smart Networking Core Technology Development and B0190-15-2012: SDN/NFV Open-Source Software Core Module/Function Development).

Digital Object Identifier: 10.1109/MCOM.2017.1600170

## Abstract

Software-defined networking is moving from its initial deployment in small-scale data center networks to large-scale carrier-grade networks. In such environments, high availability and scalability are two of the most prominent issues, and thus extensive work is ongoing. In this article, we first review the state of the art on high availability and scalability issues in SDN and investigate relevant open source activities. In particular, two well-known open source projects, OpenDaylight (ODL) and Open Network Operating System (ONOS), are analyzed in terms of high availability (i.e., network state database replication/synchronization and controller failover mechanisms) and scalability (i.e., network state database partition/distribution and controller assignment mechanisms) issues. We also present experimental results on the flow rule installation/read throughput and the failover time upon a controller failure in ONOS and ODL, and identify open research challenges.

## Introduction

Software-defined networking (SDN) is an emerging paradigm that can overcome the limitations in the current network infrastructure. The key idea of SDN is to separate the network control logic from the underlying devices that forward the traffic, and to provide the ability to program the network by means of a logically centralized controller [1]. The centralized SDN controller can easily obtain a global network view, and the performance of a network service can be optimized based on the global network view. Therefore, SDN brings many benefits such as efficient control of network traffic, reduced management cost, and rapid service deployment.

The initial concept of SDN was introduced by the Security Architecture for Enterprise Network (SANE) project [2] of the National Science Foundation (NSF) of the United States in which all routing and access control decisions within enterprise networks are made by a logically centralized server. As a practical instantiation of the SANE project, the Ethane project [3] was introduced and designed a more practical network controller. Based on the success of the Ethane project, a well-known southbound protocol, OpenFlow [4], for communications between the centralized controller and networking devices was devised. In 2011, for more systematic specification, development, and commercialization for OpenFlow, the Open Networking Foundation (ONF) was launched. Until today, various standardization organizations such as the Internet Engineering Task Force (IETF), Internet Research Task Force (IRTF), and International Telecommunication Union Telecommunication Standardization Sector (ITU-T) are working on the standardization related to SDN technologies.

While SDN was mostly targeted at data center networks or campus networks in its initial phase, SDN technologies are evolving toward SDN 2.0, which is targeted at carrier-grade networks or service providers' networks. Given the mission-critical and large-scale nature of carrier-grade networks, the control plane of SDN should be designed in a highly available and scalable manner. In this context, constructing the control plane with a single SDN controller can cause the following problems:

- A single SDN controller can become a single point of failure.
- The size of networks that can be handled by a single SDN controller is limited.

Therefore, more than one SDN controllers should be managed as a cluster, and network services/data provided by a single controller should be replicated across the cluster for high availability (HA). At the same time, for high scalability (HS), workloads should be fairly distributed across the cluster. To address these HA and HS issues, several works have been conducted in the literature, and open source communities are very active in developing highly available and scalable SDN controllers.

In this article, we first review the state of the art on HA/HS issues in SDN and survey relevant open source activities. In particular, two well-known open source projects, OpenDaylight (ODL) and Open Network Operating System (ONOS), are analyzed in terms of the HA/HS issues. We also carried out an experimental study for ONOS and ODL to show the flow rule installation/read throughput depending on the

*Dongeun Suh, Seokwon Jang, Sol Han, Sangheon Pack, and Myung-Sup Kim are with Korea University;*
*Taehong Kim was with the Electronics and Telecommunications Research Institute (ETRI), Korea, and is now with Chungbuk National University;*
*Chang-Gyu Lim is with the Electronics and Telecommunications Research Institute.*

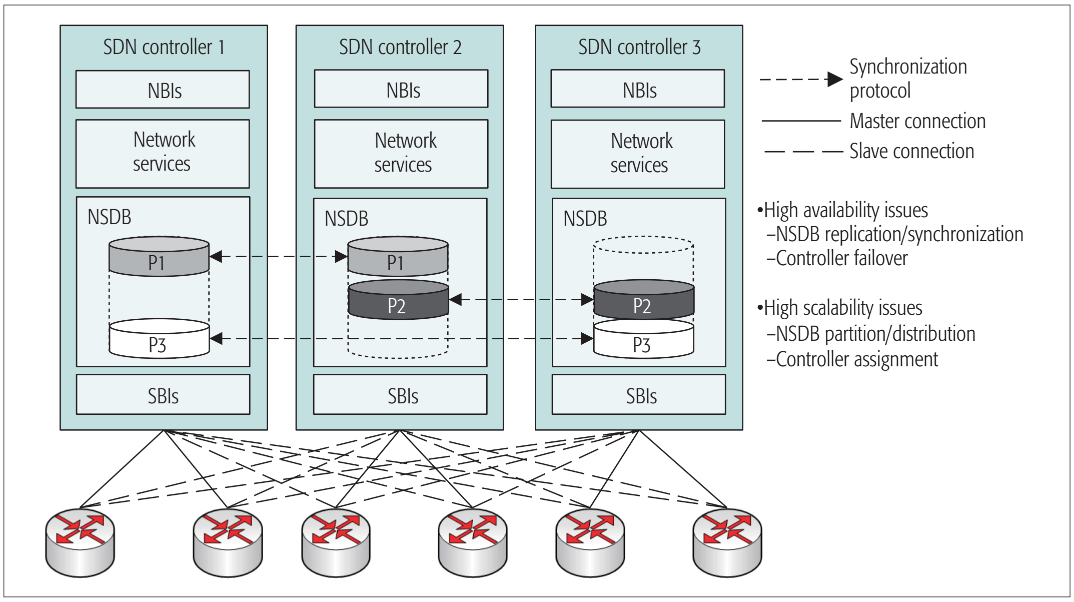

**Figure 1.** A general clustering architecture with three synchronized SDN controllers.

By a partitioning strategy, NSDB is divided into multiple partitions. Each NSDB partition can have multiple replicas for high availability, and replicas are distributed across multiple controllers for high scalability. As partitioning and distribution strategies can affect scalability, it should be carefully designed.

cluster size and the failover time upon a controller failure.

The remainder of this article is organized as follows. In the next section, key issues for high availability and scalability in SDN are identified, and relevant works are summarized. After that, how to address those issues in ODL and ONOS are discussed, and experimental results on their performance are given. Finally, this article concludes with open challenges.

## High Availability and Scalability Issues in SDN

As mentioned before, in order to construct a highly available and scalable control plane, multiple SDN controllers should be managed as a cluster. Figure 1 shows a general clustering architecture with three synchronized SDN controllers. An identical set of network services (e.g., forwarding service, network access control, etc.) are running in the controllers while their network states are stored in the distributed network state database (NSDB).

In Fig. 1, database partition/distribution and replication/synchronization techniques are deployed to the NSDB, which are well-known techniques for high scalability and availability, respectively, in a distributed database. In order to distribute data access load among controllers, the NSDB is logically partitioned into three partitions (i.e., P1, P2, P3), while replicas of partitions are fairly distributed across the cluster. Also, to cope with a controller failure, each partition is replicated into two replicas, and synchronization among the replicas is supported to maintain consistency.

Meanwhile, a master/slave model is leveraged for controller-to-device connections. That is, a device in the data plane establishes multiple connections toward controllers (i.e., master/slave connections) where a controller who has a master connection of the device is only permitted to control the device. Upon a controller failure, one of the slave connections becomes a new master connection. Also, for load balancing, each controller is assigned a subset of master connections of devices.

In such environments, four technical issues on high availability and scalability can be identified:
1. How to partition the NSDB and distribute replicas of NSDB partitions
2. How to replicate NSDB partitions in a consistent manner
3. How to recover master connections from a controller failure
4. How to assign master/slave connections for devices

Note that 1 and 4 are related to HS, while 2 and 3 are related to HA. In the following, we elaborate on each issue and summarize existing works related to the corresponding issue.

### Network State Database Partition

By a partitioning strategy, the NSDB is divided into multiple partitions. Each NSDB partition can have multiple replicas for high availability, and replicas are distributed across multiple controllers for high scalability. As partitioning and distribution strategies can affect scalability, it should be carefully designed.

Özsu *et al.* [5] presented three basic partitioning strategies in the relational database: round-robin, hash, and range partitioning. In round-robin partitioning, with $n$ partitions, the $i$th tuple in insertion order is assigned to partition $k = (i \bmod n)$. Hash partitioning applies a hash function to some attributes that yield the partition number. Range partitioning distributes tuples based on the value intervals of some attributes. Also, Özsu *et al.* [5] introduced a general fragment distribution model, which minimizes the total cost of query processing and storage on each site under the constraints of query response time and storage/query processing capacities of sites.

Krishnamurthy *et al.* [6] investigated the dependency between the application state partition and the devices. They derived the optimal assignment of switches and state partitions to distributed controllers that minimizes inter-controller communications.

In a large-scale network consisting of a number of devices, if only a few controllers act as master, the controllers might be overloaded and result in performance degradation. Therefore, the master controller should be carefully assigned for each device so that the load from the devices can be fairly distributed.

| | NSDB partition (granularity/distribution) | NSDB synchronization (protocol/consistency) | Controller failover | Controller assignment |
|---|---|---|---|---|
| ONOS EventuallyConsistentMap | Store/NA | Anti-entropy protocol/ weaker consistency | Master/slave | Same number of master connections per controller |
| ONOS ConsistentMap | Map entry/hash-based | Raft protocol/strong consistency | | |
| ODL DistributedDataStore | YANG module/ administrator-defined | Raft protocol/strong consistency | Master/slave | Same number of master connections per controller |

**Table 1.** High availability and scalability approaches in ONOS and ODL.

### Network State Database Synchronization

In a database field, synchronization strategies are used to provide consistency between replicas and can be classified into two types:
1. Synchronization strategy with strong consistency, which guarantees all replicas to return the same value when queried with an object
2. Synchronization strategy with eventual consistency, which guarantees that if no new updates are made to the object, eventually all accesses return the last updated value [5]

Meanwhile, strong consistency can only be achieved at the cost of additional latency, and different degrees of consistency can be considered. Thus, the synchronization strategy among replicas should be carefully designed.

Ongaro *et al.* [7] proposed a synchronization strategy with strong consistency, called the Raft consensus algorithm, in which all read/write requests can only be handled by a unique leader replica elected from among candidate replicas, and the read/write requests on any replicas are forwarded to the leader to be processed. For processing of write requests, the agreement among the replicas is mandatory to guarantee strong consistency. Also, in order to ensure that the leader replica is alive, the leader replica periodically sends Raft heart-beat messages to the follower replicas. If one of the follower replicas does not receive any response from the leader replica for a pre-defined election timeout, it requests a new leader election, and the replica with the most votes is elected as a new leader replica.

Botelho *et al.* [8] proposed a novel SDN architecture that focuses on highly available and strongly consistent data storage by using state-of-the-art consistent replication techniques. Botelho *et al.* [9] also developed a fault-tolerant controller architecture with a data store based on a replicated state machine and a lease management algorithm selecting a master controller for fault-tolerant SDNs.

### Controller Failover

In OpenFlow 1.2 or higher, multiple controllers for a single device are allowed for reliability, and a device maintains one of the following roles for each controller: equal, slave, and master. A device sends all OpenFlow asynchronous messages to its master controller and accepts OpenFlow controller-to-switch messages from its master controller. On the other hand, a device does not send any asynchronous messages to its slave controller and allows read-only access for it. Similar to the master controller, the equal controller has full access to the device.

The master/slave connection management is responsible for assigning new master controllers for orphan devices (i.e., devices that have lost their connections with their master controllers) while satisfying the constraint of at most one master controller. By providing such a mechanism, the number of dropped asynchronous messages from the orphan devices can be minimized.

Obadia *et al.* [10] proposed two controller failover strategies in which active neighbor controllers take over the control of orphan OpenFlow switches:
1. A Greedy strategy where neighbor controllers take over orphan switches from which they can receive messages
2. A pre-partitioning approach where neighbor controllers proactively exchange information with each other on which switches to take over upon a controller failure

### Controller Assignment

Master/slave connection management is responsible for coordination of master connections of devices. In a large-scale network consisting of a number of devices, if only a few controllers act as masters, the controllers might be overloaded, resulting in performance degradation. Therefore, the master controller should be carefully assigned for each device so that the load from the devices can be fairly distributed.

Dixit *et al.* [11] revealed that a static mapping between a network device and a controller can result in lack of dynamic load adaptation capability and proposed a switch migration protocol that can dynamically expand or shrink the controller pool depending on the traffic condition.

## Open Source Approach for High Availability and Scalability: ONOS vs. ODL

The development of SDN controllers is led by open source communities such as ONOS and ODL, and high availability and scalability are two important issues in ONOS and ODL. In this section, we briefly introduce ONOS and ODL, and explain how to address the aforementioned issues in ONOS and ODL. Key comparison results are summarized in Table 1.

### High Availability and Scalability in ONOS

Figure 2 shows an ONOS clustering architecture that consists of an identical set of network services running in each ONOS instance (only shown for ONOS1 for simplicity) and a middleware component, called Distributed Core, that

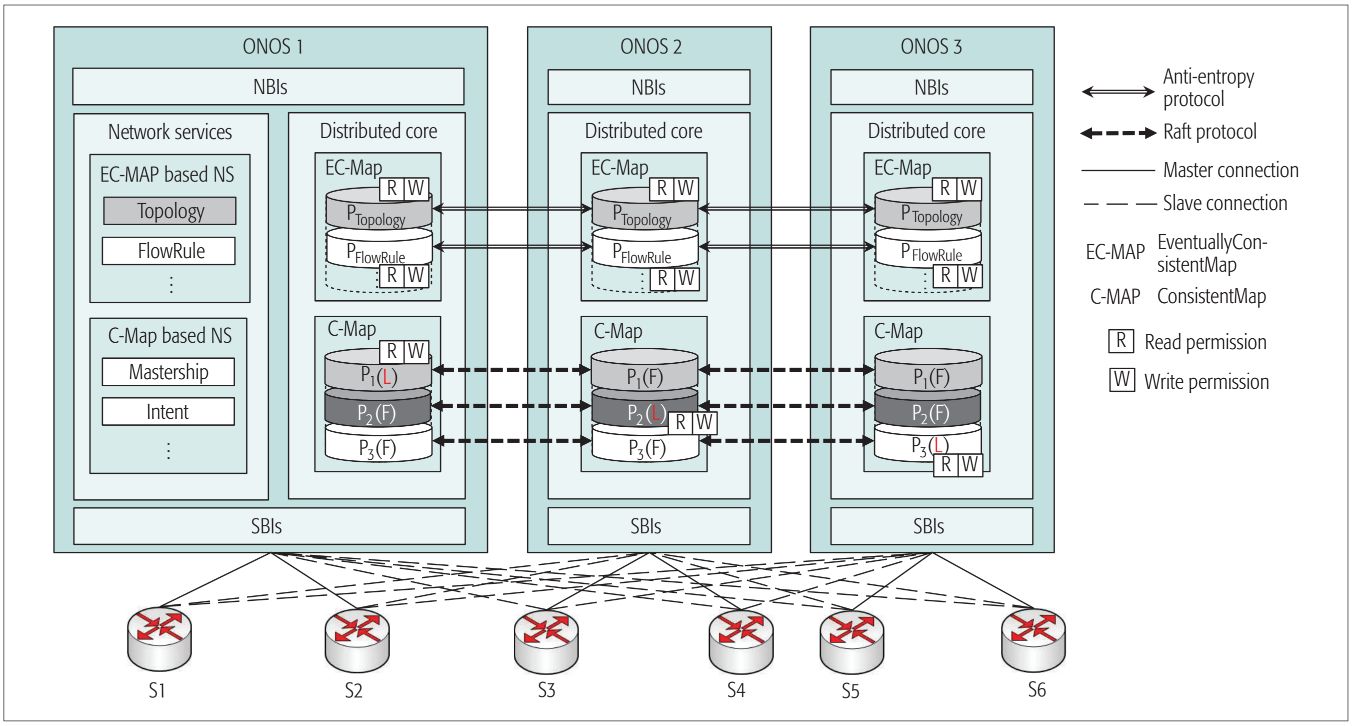

**Figure 2.** An example of NSDB partitioning/synchronization in ONOS.

manages distributed operations across the cluster and provides two types of NSDBs with different partitioning and synchronization strategies:
1) EventuallyConsistentMap
2) ConsistentMap
Each network service implements its own data storage called Store by means of the two types of NSDBs and accordingly can be classified by which type of NSDB it uses. As shown in Fig. 2, topology and flow rule services are based on EventuallyConsistentMap, while mastership and intent services are based on ConsistentMap. Also, ONOS allows each device to have multiple connections to multiple ONOS controllers, and the master/slave connection management is provided for load balancing and controller failover.

In order to detect a controller failure, ONOS leverages a φ-accrual failure detector [12] where controllers exchange heartbeat messages periodically to keep track of the suspicion level of failure φ for each controller. Each ONOS controller calculates the values of φ for other controllers as $\varphi = -\log10(1 - F(t))$ where $F(t)$ is the cumulative distribution function of a normal distribution with mean and standard deviation estimated from historical heartbeat inter-arrival times $t$. If a value of φ for an ONOS controller is greater than a pre-defined threshold Φ, the controller is considered as failed.

**Network State Database Partition:** EventuallyConsistentMap is partitioned into $S$ partitions where $S$ denotes the number of network services, and each partition contains data of each network service (i.e., Store). As shown in Fig. 2, $P_{\text{Topology}}$ and $P_{\text{FlowRule}}$ contain data of topology and flow rule services, respectively. Meanwhile, all partitions of EventuallyConsistentMap are fully replicated into all controller instances joining the cluster.

On the other hand, ConsistentMap is partitioned into $n$ partitions where $n$ is configurable by the administrator and set to the number of controllers in the cluster by default. Data to be contained in each partition is determined by a hash value of each ConsistentMap entry's key where the hash range is $[1, N]$. For example, in Fig. 2, $P_1$ contains ConsistentMap entries whose hash values are 1. For ConsistentMap, each partition has $R$ replicas where $R$ is a configurable parameter, and each replica is assigned to the controller that has the least number of replicas. Figure 2 shows a case when $R = 3$.

**Network State Database Synchronization:** For EventuallyConsistentMap, replicas of each partition are synchronized based on the anti-entropy protocol [14]; it provides weaker consistency guarantee in return for superior read/write performance. All replicas of a partition of EventuallyConsistentMap can handle read/write requests. Specifically, read requests are handled only by the local replica, whereas write requests are handled by the local replica first and the updates are subsequently propagated to other replicas. In order to resolve write conflict and ensure replica convergence, upon receiving an update event for an EventuallyConsistentMap entry, a replica assigns the logical timestamp to the update. Then the update is committed into the EventuallyConsistentMap entry and, in parallel, broadcasted along with the timestamp to other replicas. Upon receiving the broadcasted update event, each replica checks if it has a more recent update for the entry. If the received timestamp is older, it discards the update. Otherwise, the update is committed into its EventuallyConsistentMap entry. By doing so, the system state across all replicas eventually converges to the correct state.

Also, in order to promptly synchronize a replica of a newly joining or restarted controller, at fixed intervals, each replica randomly selects

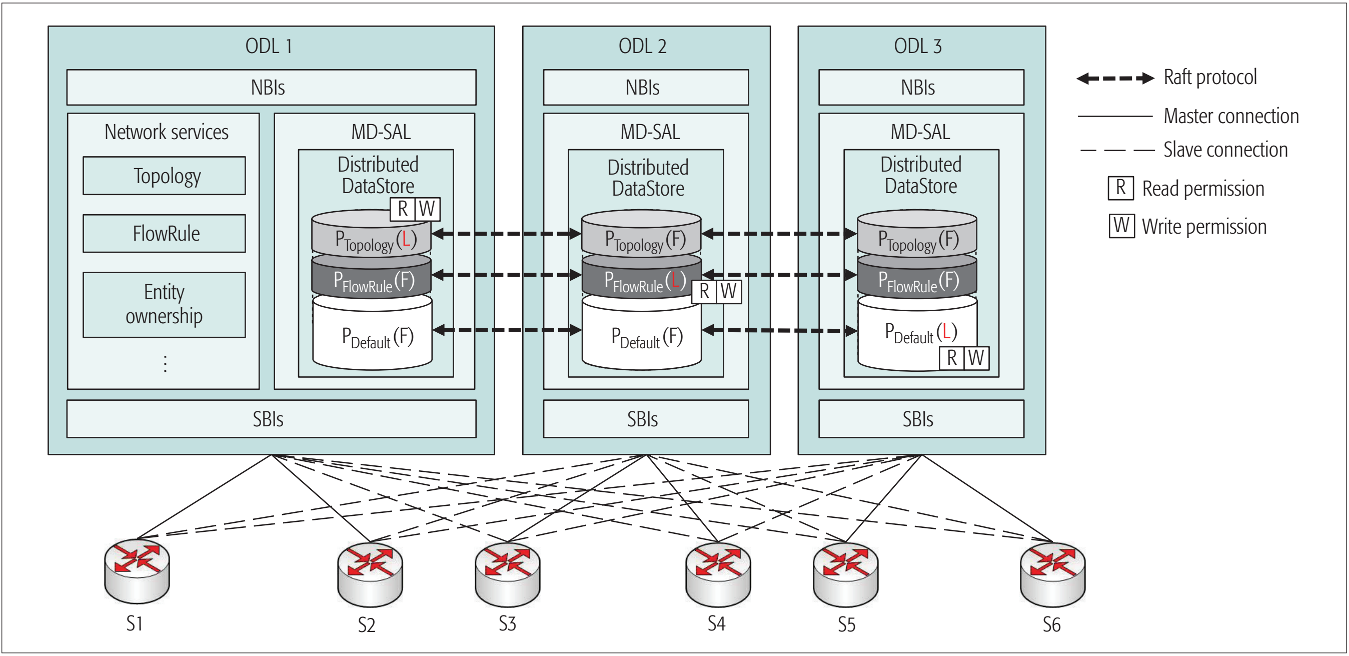

**Figure 3.** An example of NSDB partitioning/synchronization in ODL.

another replica, and they both synchronize their states. If one replica is aware of more recent EventuallyConsistentMap entries that the another replica does not have, they exchange the entries. For example, in Fig. 2, consider that a topology service running in ONOS3 received a topology update event from S6 and issued a write request to the topology state. The write request is committed into the local replica (i.e., $P_{Topology}$ in ONOS3) immediately and, in parallel, propagated to other replicas. Meanwhile, before the write request is committed into $P_{Topology}$ in ONOS1 and ONOS2, read requests to $P_{Topology}$ in ONOS1 and ONOS2 may observe stale topology state.

For ConsistentMap, replicas of each partition are synchronized based on the Raft protocol [7], which provides strong consistency at the cost of inferior read/write performance. For example, in Fig. 2, consider that an intent service running in ONOS1 has issued a write request to the ConsistentMap entry contained in $P_3$. Since the local replica is a follower (i.e., $P_3$(F) in ONOS1), the request cannot be handled locally and should be forwarded to the leader replica (i.e., $P_3$(L) in ONOS3). Then, after obtaining agreements among the replicas, the leader replica commits the write request.

**Controller Failover:** Upon an ONOS controller failure, other ONOS controllers in the cluster detect the failure by the failure detector, and re-assign a new master controller for orphan devices. The newly elected master controller for each device sends a role request message to the device to set its role to the device as master, and if successful, it receives a role reply message from the device. As a result, all the orphan devices can recover their master connections.

**Controller Assignment:** When a new device is connected to multiple ONOS controllers, its master controller is set to the controller that has the smallest number of master connections. By doing so, the number of devices that each controller serves as the master becomes balanced.

## High Availability and Scalability in ODL

Figure 3 shows an ODL clustering architecture that consists of an identical set of network services running in each ODL instance (only shown for ODL1 for simplicity) and a middleware component, called the model driven-service abstraction layer (i.e., MD-SAL), that manages distributed operations across the cluster and provides an NSDB called DistributedDataStore. Different from ONOS, each network service in ODL models its data as a form of the YANG module [13] where YANG is a data modeling language. Based on the YANG modules, DistributedDataStore is constructed to store data of network services. Also, similar to ONOS, ODL provides the master/slave connection management and uses a φ-accrual failure detector.

**Network State Database Partition:** In ODL, the administrator partitions DistributedDataStore into several partitions and selects which YANG module is to be contained in the partitions. There is one special partition called Default Shard, which contains all data except the data defined by the selected YANG modules by the administrator. As shown in Fig. 3, YANG modules of topology and flow rules can be selected by the administrator, and DistributedDataStore can be partitioned into three partitions accordingly. Each partition is replicated into $R$ replicas where $R$ is configurable by the administrator. Figure 3 shows a case when $R$ = 3. Similar to ONOS, each replica is assigned to the controller with the least number of replicas.

**Network State Database Synchronization:** As in ConsistentMap in ONOS, ODL uses the Raft protocol [7] for synchronization between replicas of a partition. Different from ONOS, all network services in ODL are provided with the Raft protocol for synchronization between their replicas. For example, in Fig. 3, consider that a topology service running in ODL3 received a topology update event from S6 and issued a write request to the topology state. Since the local replica is a follower (i.e., $P_{Topology}$(F) in ODL3), the request cannot

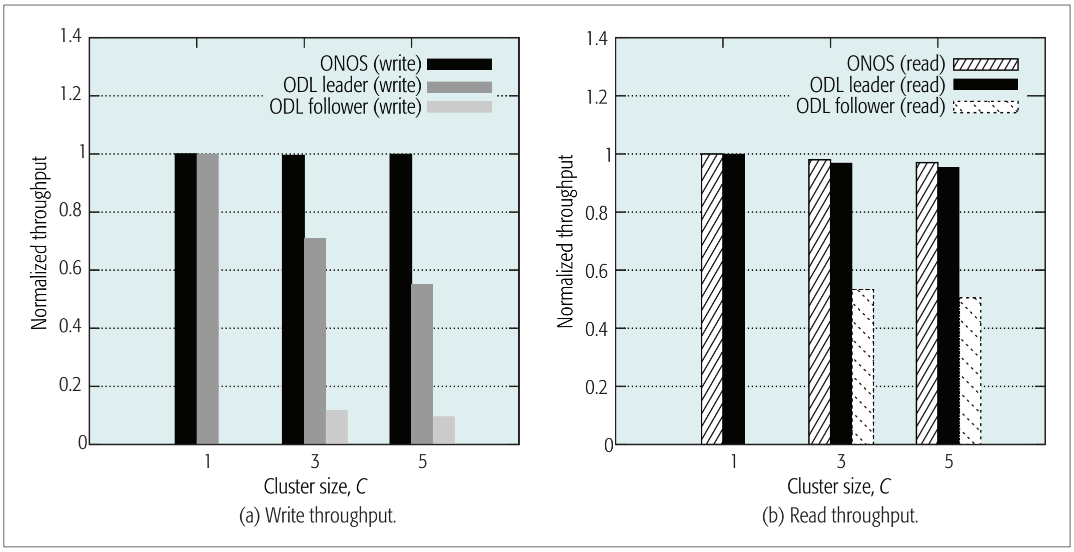

**Figure 4.** Normalized flow rule installation/read throughput: ONOS vs. ODL.

The newly elected master controller for each device sends a role request message to the device to set its role to the device as master, and if succeeded, it is replied with a role reply message from the device. As a result, all the orphan devices can recover their master connections.

be handled locally and thus should be forwarded to the leader replica (i.e., $P_{\text{Topology}}(L)$ in ODL1); upon obtaining agreements among replicas, the leader replica commits the write request. Therefore, although the consistency between topology state replicas is guaranteed all the time in ODL, read/write performances can be degraded.

**Controller Failover and Controller Assignment:** The controller failover and controller assignment mechanisms in ODL are similar to those of ONOS; therefore, we have omitted the corresponding descriptions.

## Experimental Results

For comparative study, we evaluate the performance of ONOS and ODL in terms of flow rule installation/read throughput and controller failover time. We run each ODL controller in 6 GB RAM and a 2 CPU core virtual machine (VM) with Ubuntu 14.04.2 LTS and each ONOS controller in 6 GB RAM and a 2 CPU core VM with CentOS 6.7, respectively. In terms of version of ODL and ONOS, we use ODL lithium-SR3 distribution and ONOS-1.4 (Emu) distribution. Each experiment is repeatedly carried out to obtain reliable sample values, and the results are obtained by averaging the sample values.

To evaluate the flow rule installation throughput, we run $C$ ONOS/ODL controllers, assign one ONOS/ODL controller as a master controller of nine devices, and install 500 flow rules per device through the master controller. Also, the partition that contains the flow rule state is fully replicated into all controllers in ONOS/ODL. For ODL, flow rules are generated, contained in HTTP POST messages, and then transmitted through the northbound REST application programming interface (API) of the master controller to add/delete the bulk of the flow rules into the Inventory Shard that contains flow rules. After that, the OpenFlow controller service in ODL is notified of the data change in Inventory Shard and installs the flow rules to OpenFlow switches. On the other hand, a flow rule installation request tool is used in ONOS.[1] Specifically, the tool requests installation of flow rules to the flow rule throughput test application running in the master controller. After that, the test application creates flow rules randomly and writes the flow rules into the local FlowRule Store, which is based on the EventuallyConsistentMap. As a sequel, the OpenFlow controller service in ONOS installs the flow rules to OpenFlow switches. Since the flow rule installation procedures of ONOS and ODL are different from each other, flow rule installation throughput values of ONOS and ODL are not directly comparable. Therefore, we consider the normalized flow rule installation throughput for ONOS and ODL where the flow rule installation throughput values of ONOS and ODL are normalized by the flow rule installation throughput values when $C$ is 1 for ONOS and ODL, respectively.[2]

Figure 4a shows the normalized flow rule installation throughput depending on the number of controllers in the cluster, $C$. For ONOS, it can be seen that the throughput is rarely affected by $C$. This is because since all replicas can handle read/write requests, upon receiving flow rule installation requests, the master controller updates its local replica first. On the contrary, in ODL, two different results are obtained when:

1. The master controller contains a leader replica.
2. The master controller contains a follower replica.

For case 1, agreement among the follower replicas is mandatory before committing the flow rules into the leader replica. Consequently, the latency for committing flow rules increases and the throughput decreases with the increase of $C$. Also, in case 2, degraded throughput is observed as $C$ increases due to the increased commitment latency. Moreover, case 2 shows drastically reduced throughput compared to case 1. In case 2, when the master controller receives flow rule installation requests, it forwards the requests to the controller that contains the leader replica. Only after the flow rules are committed into the leader replica, the master controller is notified with the flow rule changes remotely. This forwarding of flow rule installation requests and remote flow rule change notifications cause additional latency; therefore, case 2 shows drastically reduced throughput.

Meanwhile, there is a trade-off between

[1] The northbound REST API for the bulk flow rule installation is under development and unavailable in ONOS-1.4 (Emu) distribution.

[2] The flow rule installation throughput values of ONOS and ODL when $C$ is 1 are 13,436.7 flows/s and 4672.4 flow/s, respectively.

As ODL provides strong consistency for flow rules, the flow installation throughput can be degraded compared to ONOS which provides eventual consistency for flow rules. However, the eventual consistency for flow rules in ONOS may potentially present a temporal inconsistency and cause undesired behavior of network services that subscribe flow rules.

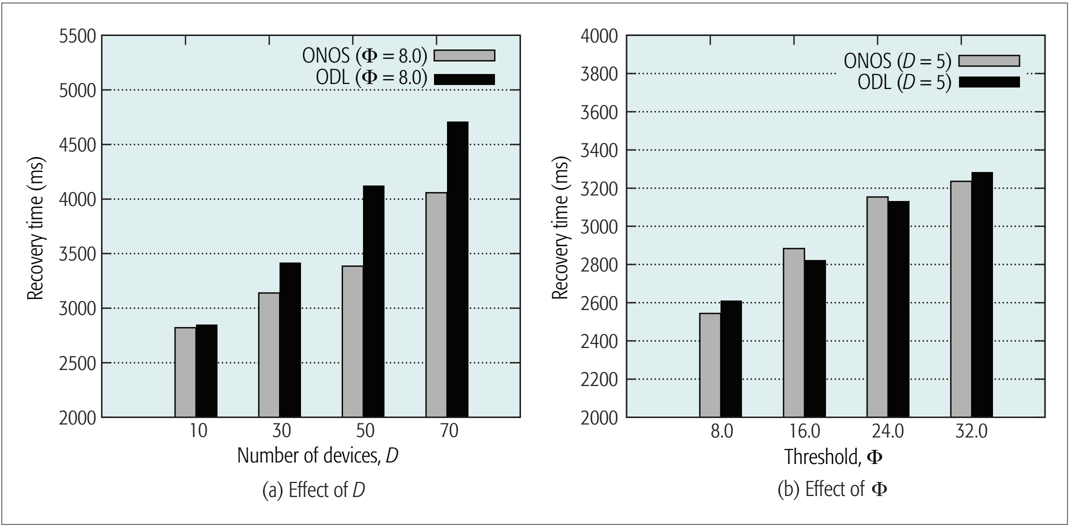

**Figure 5.** Controller failover time: ONOS vs. ODL.

ONOS and ODL in terms of the flow rule consistency and the flow rule installation throughput. As ODL provides strong consistency for flow rules, the flow installation throughput can be degraded compared to ONOS, which provides eventual consistency for flow rules. However, the eventual consistency for flow rules in ONOS may potentially present a temporal inconsistency and cause undesired behavior of network services that subscribe flow rules.

To evaluate the flow rule read throughput, we installed 3000 randomly generated flow rules into $C$ ONOS/ODL controllers and transmitted a flow rule read request through the northbound REST API of a target controller. Upon receiving the read request, the target controller replies with the requested flow rule. Similar to the flow rule installation experiment, the flow rule read throughput values of ONOS and ODL are normalized by the flow rule read throughput values when $C$ is 1.[3]

As shown in Fig. 4b, the throughput of ONOS is rarely affected by $C$ for to the same reason as in the flow rule installation throughput experiment. On the other hand, the two cases described in the flow rule installation experiment are considered in ODL. For case 1, the flow rule read requests do not require any agreement among the replicas. Therefore, a flow read request can be locally processed by the leader replica, and the throughput remains constant even with the increase of $C$. Also, in case 2, it can be seen that the throughput is rarely affected by $C$ as the flow rule read requests do not require any agreement among the replicas. Meanwhile, case 2 shows drastically reduced read throughput compared to case 1, which can be explained by the same reason as in the flow rule installation experiment in ODL.

In the controller failover experiment, we run three ONOS/ODL controllers and select one ONOS/ODL controller as a master controller of $D$ devices. For the failover scenario, the master controller is intentionally turned down, and the elapsed time from the last heartbeat message of the failure detector running in the master controller to the time when a role reply message from the last orphan device is received by a new master controller is measured.

Figure 5a shows the effect of the number of devices, $D$, on the failover time when $\Phi = 8$. For both ONOS and ODL, as $D$ increases, the number of orphan devices upon a failure increases and the number of role request messages to be sent increases. As a result, the failover time increases with the increase of $D$ as shown in Fig. 5a.

Figure 5b demonstrates the effect of $\Phi$ on the failover time when $D = 5$. It can be seen that the failover time increases as $\Phi$ increases both in ONOS and ODL. This can be explained as follows. As $\Phi$ increases, the cluster in ODL and ONOS becomes more conservative in determining a controller failure. Therefore, the elapsed time between the failure event and the failure detection event is incremental to $\Phi$, and the failover time for ONOS and ODL increases accordingly.

## Conclusion

In this article, we discuss high availability and scalability issues in SDN, and analyze ONOS and ODL approaches. Experimental results demonstrate that:
1. The flow rule installation throughput of ODL is significantly affected by the cluster size.
2. There is a trade-off between ONOS and ODL in terms of the flow rule consistency and the flow rule installation throughput.
3. The controller failover time is dependent on the number of devices and the failure detection threshold.

As open challenge:
1. There is a trade-off between inconsistency of network states and performance of network services.
2. The controller assignment problem in large-scale WAN environments must be addressed, where latencies between controllers and switches are significant.
3. Stability analysis in large-scale SDNs with a few tens or hundreds of controllers in a cluster should be further investigated, and ONOS and ODL will evolve to address these challenges.

## References

[1] D. Kreutz *et al.*, "Software-Defined Networking: A Comprehensive Survey," *Proc. IEEE*, vol. 103, no. 1, Jan. 2015, pp. 14–76.

[3] The read throughput values of ONOS/ODL when $C$ is 1 are 13,946.7 and 20,813.8 flows/s, respectively.

[2] Security Architecture for Enterprise Network (SANE) project. http://yuba.stanford.edu/sane/.

[3] M. Casado *et al.*, "Ethane: Taking Control of the Enterprise," *ACM SIGCOMM Comp. Commun. Review*, vol. 37, no. 4, Oct. 2007, pp. 1–12.

[4] N. McKeown *et al.*, "Openflow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Comp. Commun. Review*, vol. 38, no. 2, Apr. 2008, pp. 69–74.

[5] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems*, Prentice-Hall, 2007.

[6] A. Krishnamurthy, S. Chandrabose, and A. Gember-Jacobson, "Pratyaastha: An Efficient Elastic Distributed SDN Control Plane," *Proc. ACM SIGCOMM Wksp. Hot Topics in Software Defined Networking 2014*, Chicago, IL, Aug. 2014.

[7] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," *Proc. USENIX Annual Technical Conf. 2014*, Philadelphia, PA, June 2014.

[8] F. Botelho *et al.*, "On the Feasibility of a Consistent and Fault-Tolerant Data Store for SDNs," *Proc. Euro. Wksp. Software Defined Networks 2013*, Berlin, Germany, Oct. 2013.

[9] F. Botelho *et al.*, "On the Design of Practical Fault-Tolerant SDN Controllers," *Proc. Euro. Wksp. Software Defined Networks 2014*, Budapest, Hungary, Sept. 2014.

[10] M. Obadia *et al.*, "Failover Mechanisms for Distributed SDN Controllers," *Proc. IEEE Int'l. Wksp. Network of the Future 2014*, Paris, France, Dec. 2014.

[11] A. Dixit *et al.*, "Towards an Elastic Distributed SDN Controller," *Proc. ACM Wksp. Hot Topics in Software-Defined Networking 2013*, Hong Kong, Aug. 2013.

[12] N. Hayashibara *et al.*, "The φ accrual Failure Detector," *Proc. IEEE Int'l. Symp. Reliable Distributed Systems 2004*, Florianpolis, Brazil, Oct. 2004.

[13] M. Bjorklund, "YANG — A Data Modeling Language for the Network Configuration Protocol (NETCONF)," IETF RFC 6020, Oct. 2010.

[14] A. Demers *et al.*, "Epidemic Algorithms for Replicated Database Maintenance," *Proc. ACM Symp. Principles of Distributed Computing 1987*, Vancouver, BC, Aug. 1987.

## Biographies

Dongeun Suh [M] (fever1989@korea.ac.kr) received his B.S. degrees from Korea University, Seoul, in 2012. He is currently a Ph.D. student in the School of Electrical Engineering, Korea University. From 2012 to 2016, he received a scholarship from Samsung Electronics. His research interests include SDN/NFV/DTN and multimedia streaming.

Seokwon Jang (imsoboy2@korea.ac.kr) received his B.S. degree from Korea University in 2015. He is currently an M.S. and Ph.D. integrated course student in the School of Electrical Engineering, Korea University. His research interests include SDN/NFV, future Internet, and programmable networking.

Sol Han (hs1087@korea.ac.kr) received his B.S. degree from Korea University in 2015. He is currently an M.S. and Ph.D. integrated course student in the School of Electrical Engineering, Korea University. His research interests include SDN/NFV, future Internet, and programmable networking.

Sangheon Pack [SM] (shpack@korea.ac.kr) received his B.S. and Ph.D. degrees from Seoul National University, Korea, in 2000 and 2005, respectively, both in computer engineering. In 2007, he joined the faculty of Korea University, where he is currently a professor in the School of Electrical Engineering. He was the recipient of the Korean Institute of Communications and Information Sciences (KICS) Haedong Young Scholar Award 2013 and the IEEE ComSoc APB Outstanding Young Researcher Award in 2009. His research interests include future Internet, softwarized networking (SDN/NFV), mobility management, and mobile cloud networking/edge computing.

Myung-Sup Kim (tmskim@korea.ac.kr) received his B.S., M.S., and Ph.D. degrees in computer science and engineering from POSTECH, Korea, in 1998, 2000, and 2004, respectively. He joined Korea University in 2006, where he is currently a professor in the Department of Computer and Information Science. His research interests include Internet traffic monitoring and analysis, SDN/NFV, and Internet security.

Taehong Kim (taehongkim@cbnu.ac.kr) received his Ph.D. degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST) in 2012. He has been an assistant professor with the School of Information and Communication Engineering, Chungbuk National University, Korea, since March 2016. He worked as a research staff member with Samsung Electronics and ETRI from May 2012 to February 2016. His research interests include wireless sensor networks, the Internet of Things, and SDN/NFV.

Chang-Gyu Lim (human@etri.re.kr) is a senior engineer of SDN Research Section, ETRI, Korea. He received his Master's degree at KAIST in 2002. His key research interests are: future Internet, software defined networking, and transport networks.