Realizing Flat Multi-Zone Multi-Controller Software-Defined Networks using Zenoh

Federico Giarré
DISI, University of Trento, Italy
federico.giarre@gmail.com

Luca Cominardi
ZettaScale Technology, France
luca@zettascale.tech

Paolo Casari
DISI, University of Trento, Italy
paolo.casari@unitn.it

***Abstract*—Software Defined Networks (SDNs) are becoming an increasingly important paradigm in modern networking. By making it possible to detach the control plane from network devices and to instruct those devices via a high-level programming language, SDNs yield a number of administration-level improvements, such as better policy-driven network management and easier support for multi-tenancy. However, moving the complexity of network administration to the controllers entails additional challenges, such as scalability, support for different controller network topologies, and efficient maintenance of network state information. In this paper, we consider a distributed and decentralized approach to the SDN paradigm, targeting zero-configuration scalability and transparency through the Zenoh framework. We explore the application of Zenoh as an east/westbound protocol for SDNs and compare its performance against that of a strongly coupled HTTP-based alternative. Our results show that Zenoh's support for distributed queries enables our SDN architecture to effectively scale to hundreds of SDN controllers with zero configuration overhead, while supporting high east/westbound query throughput.**

***Index Terms*—Software defined networking; flat controller hierarchy; Zenoh; query consolidation; performance evaluation**

I. Introduction and related work

Software Defined Networks (SDNs) have gained increasing popularity over time, mainly due to the advantages that such technology provides: easier network management and monitoring, as well as full control over the packet forwarding process. In particular the latter, if implemented correctly, yields greater flexibility, scalability, and efficiency to the network. In an SDN, the control plane of the network devices (e.g., the switches) is detached and moved to a different device, called the controller. The controller is in charge of serving forwarding information to the devices of the network in order to enable end-to-end communications.

For any level-2 network, one controller is sufficient to operate the network. However, the presence of additional distributed controllers enables network administrators to implement such paradigms as multi-tenancy, while decreasing delays and bandwidth usage within the network. Yet, scaling a distributed network of SDN controllers implies several challenges. First, for every new controller connected to an existing SDN, the controllers already in place typically need to i) update their configuration and ii) to propagate such changes to their peers by exchanging update messages, e.g., via efficient publish/subscribe message passing architectures. Second, to operate distributed SDNs efficiently, every controller should have a complete (or anyway very broad) view of the whole network, so that effective forwarding rules can be determined.

Achieving the above requires the implementation of a logically and physically distributed SDN framework. In fact, distributing the knowledge of the network between multiple controllers reduces the state that every controller needs to handle, and enables effective scaling. While one immediate solution would be to arrange controllers hierarchically [1], managing the hierarchy is challenging if controllers can be added or removed over time. Alternative solutions such as clustering [2] attempt to improve the reliability of the SDN by grouping controllers into clusters. However, this moves the complexity from hierarchy management to cluster formation and clusterhead rotation. Conversely, a flat controller topology would enable not only smooth controller addition/removal, but also virtually infinite horizontal scaling. Yet, distributed controllers mean distributed information (e.g., device reachability, local network topologies, etc.) and bring new challenges: how to connect the controllers, and how to retrieve data from them. Mesh controller networks are a typical solution to let controllers connect and communicate [3]. Yet, such a topology requires significant heavier management as the number of controllers increases. Moreover, while information retrieval from other controllers can be achieved through standard east/westbound interfaces, we need to balance the amount of data exchanged, e.g., with the help of caching mechanisms in order to prevent network bandwidth saturation.

A. Related work on multi-controller SDN frameworks

Many multi-controller SDN frameworks exist. The survey in [4] describes the combinations of centralized/distributed control architectures (physically, logically or both), whereas [5] discusses the characteristics of different interface protocols for northbound, southbound and east/westbound interfaces. Flat structure frameworks for multi-zone multi-controller SDN are available (e.g., Onix [6], HyperFlow [7], and ONOS [8]) but they require a unified view of the network, in order to achieve reliability and to make the best forwarding decisions. Other frameworks like KANDOO [9] and ORION [10] are logically distributed, but require full (KANDOO) or hybrid (ORION) hierarchy to work. Container-based (or micro-services-based) SDN architectures [11], distribute the services of a monolithic SDN controller across the network. This allows the system to achieve greater scalability and modularity at the cost of higher latency [12] and of heavier difficult management, due to micro-service orchestration and load balancing.

B. Contribution of this work

From our analysis of the literature, we conclude that there is currently a lack of frameworks that cache only information about the network zone managed by each controller, and store little to no information about other zones. In this paper, we tackle the above challenges by proposing and evaluating a framework that realizes both a distributed, physically-flat SDN control architecture (i.e., it does not rely on a centralized controller, or a hierarchy thereof) and a distributed logical control architecture (where no controller needs complete knowledge of data locations within the network).

In the remainder of this paper we present our envisioned SDN architecture (§II), describe our use case and evaluation scenario (§III), discuss performance evaluation results (§IV) and finally draw concluding remarks in §V.

II. Envisioned SDN architecture

A. Overview and assumptions

Our working assumption in this paper is that the physical network is subdivided into zones, and that there exists one controller managing each zone. Controllers have full knowledge of their own zone (e.g., in terms of available devices, exposed resources, etc.), but ignore other parts of the network. Therefore, controllers will resort to distributed queries targeting other controllers, in order to retrieve and compute the information required to forward packets outside their own zone. The only information controllers can seek and store about other zones relate to border devices, i.e., those devices that host the physical link connecting two different zones. Controllers will rely on border devices (e.g., a switch in a simple L2 use case) to compute the path for packets to reach other zones, using these same border devices as gateways.

All distributed queries pass through east/westbound interfaces, and are mediated by the corresponding communications protocols. East/westbound interfaces enable notification messages (such as updates about device/resource changes of interested for the communicating controllers) and make it possible to retrieve the metadata required for controller to function. In this work, we employ Zenoh [13] as an east/westbound communication protocol. We provide a general overview and the required details about Zenoh in the following subsection.

B. Zenoh

Zenoh is a middleware and communication protocol specifically designed to address the challenges of applications that operate in a decentralized and heterogeneous system such as edge/fog computing [14]. Specifically, Zenoh unifies data in motion, data in-use, data at-rest and computation by blending traditional publish/subscribe patterns with geographycally distributed storage, queries and computations. In doing so, it is designed to retain good time and space efficiency for any type of data communication: peer-to-peer, brokered, or routed via a Zenoh infrastructure. Additionally, Zenoh adopts a Named-Data Networking (NDN) paradigm that provides location transparency to data communications and data storage, allowing applications to focus on actual data, rather than on their location. To that end, Zenoh natively integrates decentralized databases and supports eventual consistency for data [15] via a dedicated storage alignment mechanism reacting to system failures. Zenoh uses human-readable names to expose network resources. For instance, a controller could store a retrievable value as `controller1/value`. Distributed stored values can be queried using wildcards: e.g., `**/value` retrieves `value` from any Zenoh storage hosting it.

Zenoh assigns a timestamp to each resource to ensure eventual consistency of data. The user can hence apply consolidation policies to filter incoming responses. Three consolidation policies are available. *None* retrieves all the data samples of a given resource. Samples are neither ordered nor filtered, and any consolidation policy is delegated to the application. *Latest Value* ensures that only the newest sample of a given resource is returned when performing a query. Zenoh needs to wait to retrieve all samples firsts, and then singles out the newest sample. This is the slowest and heavier consolidation policy. *Monothonic* returns only newer samples than the last retrieved one for a given resource, and thus represents a tradeoff between *None* and *Latest Value* consolidation.

SDN controllers can take advantage of the Zenoh middleware and protocol by using Zenoh's distributed storage and query capability to i) store and expose their state and ii) retrieve the state from other controllers. Zenoh's pub/sub functionalities can then be used to notify other controllers about updates in their zone. We remark that Zenoh's NDN-based operational design is in contrast with strongly coupled and location-dependent protocol options (e.g., as realized through REST APIs relying on HTTP), which require to know the location of a resource and to establish one-to-one connections between every pair of existing devices, greatly limiting the scalability of SDNs. At the same time, Zenoh provides the SDN controllers with the required level of consistency to store and share their state.

Zenoh tackles the above complexity by interconnecting controllers and by allowing them to locate and reach any devices or resources through distributed queries, with no changes to their codebases or previous knowledge on their location. Moreover, Zenoh does not need coupled communications, and controllers can be arranged into any topology (e.g., peer-to-peer, mesh or even across different networks using the Zenoh Routers extension). For instance, if controllers A and B are directly connected, we can connect controller C only to B. With Zenoh, controller C will be able to perform queries on both controllers A and B (and vice-versa, controllers A and B will be able to query controller C). Thanks to this, connecting a new network section to the existing ones is as easy as booting up the controller and connecting the SDN controller to the Zenoh system. Other uses of Zenoh include the seamless management of topology changes, e.g., due to device relocation or collective controller reconfigurations towards some common

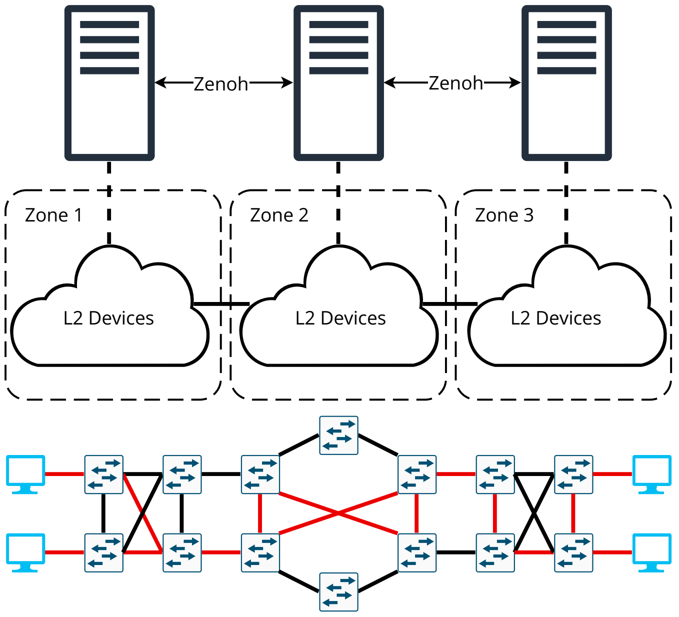

Fig. 1. (Top) Scheme of the topology of our considered use case and (bottom) example of realization of a topology including L2 devices, shown as blue cubes with crossing arrows. The devices are interconnected by either 1-Gbit/s links (red) or 1-Mbit/s links (black).

goal (e.g., ensuring a minimum amount of bandwidth on some end-to-end path). Similarly, by storing the SDN controllers' state in Zenoh decentralized storage, switches can be migrated from one controller to another, even across different zones, since data can be accessed transparently.

III. Considered use case and experimental setup

A. Use case description

The scenario we focus on is an SDN applied to a large enterprise or university campus network. These examples are representative of ever-evolving networks that can benefit from transparent addition/removal of new network sections that do not affect existing sections. For example, if a university wants to add new temporary laboratories, the controllers in other parts of the network have to account for the changes and rearrange their knowledge in order to ensure connectivity. This process can require heavy or cumbersome setup and maintenance operations, which include reverting the configuration to the *status quo*, once temporary laboratories are discontinued.

Even in the presence of an evolving topology, network devices and resources on the network should be seamlessly available and reachable. Therefore, this scenario can benefit significantly from the plug-and-play controller connection functionalities offered by Zenoh.

As a reference topology for our use case, we consider the setup of Fig. 1. In this setup, the network is subdivided into three zones, each managed by a single controller. Zones include devices and an L2 connectivity fabric. The bottom panel of Fig. 1 depicts a possible physical network topology, where zones 1 and 3 include both devices and switches,

TABLE I
Configuration of the physical and virtual testbed platforms

	Physical workstations	Controller VMs
CPU	8-core AMD Ryzen 7 5800X	4 vCPUs
Memory	32 Gb	8 Gb
OS	Ubuntu 20.04.3 LTS	Ubuntu 20.04.3 LTS

whereas zone 2 is mainly an interconnecting zone, and contains only switches. SDN controllers communicate with one another using Zenoh as their east/westbound protocol.

B. Experimental setup

We implement the above scenario using the following setup. We employ a workstation embedding an 8-core AMD Ryzen 7 5800X CPU, 32 GByte of RAM, and running on an Ubuntu 20.04.3 LTS operating system. We virtualize computing and networking resources on this workstation in order to run SDN controllers, as well as instantiate the L2 fabric and all network devices. In particular, all controllers run on virtual machines (VMs) managed by KVM, whereas we employed Mininet [16] to emulate connectivity, network devices and switches. The configuration of the physical and virtual machines are summarized in Table I.

As the SDN zone controller we chose Ryu [17]. While Ryu is a physically and logically centralized SDN framework, we recall that the controller structure is flat and distributed, and every controller manages its own zone, without having to maintain any knowledge about devices and resources available in other zones. Because the scope of each controller is limited to a single network zone, a fully centralized controller perfectly fits our purposes.

Further, we deployed two additional workstations equal to the one hosting the controller VMs and the L2 devices. This enables us to make routed tests, where different instances of Zenoh are separated by at least one physical network hop. Moreover, this prevents effecting the results by having too many SDN controllers running on virtualized resources on the same physical machine, without an actual network in between. Instead, with this setup we address the case where different controllers instances are displaced throughout different physical places (as is typical in realistic large networks), and cannot connect in a peer-to-peer manner.

We relied on two different types of processes. *Getter*s start a Zenoh session and connect to a Zenoh router, in order to perform distributed queries at the epochs determined by the user. *Queryable* processes, conversely, store some value that will be served to the Getter process upon receiving the distributed queries. Queryables also start a Zenoh session and connect to the Zenoh router at startup. Both Getters and Queryables are SDN controllers.

In particular, following the scheme in Fig. 2, we connect the three physical machines via 100-Gbit/s Ethernet links, and run one Getter on the leftmost machine, and multiple Queryables on the rightmost machine. The same machine that hosts the controllers and L2 devices also instantiates a Zenoh

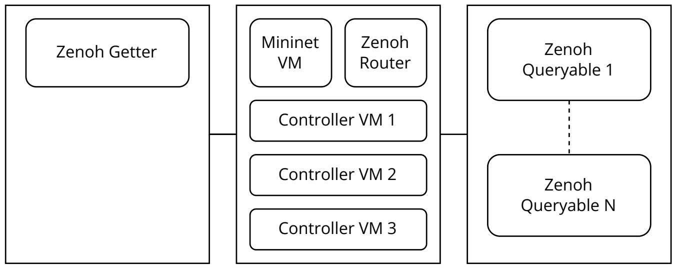

Fig. 2. Setup used for our performance evaluation, with three physical workstations connected via 100-Gbit/s Ethernet links.

router. With this setup, the controllers have to communicate outside their own virtual machine in order to retrieve data in the network. We inspected several alternative deployments of Getters and Queryables, but did not experience any significant performance differences. As a summary reference, the software packages used to implement and execute the tests are Mininet 2.3.0; Ryu 4.34; Openflow 1.3; Scapy 2.4.5; Zenoh 0.6.0-dev (clique-peers branch); Zenoh nightly for Python.

IV. Experimental evaluation

In this section we present a performance evaluation of our envisioned SDN architecture. Section IV-A investigates the delays incurred by controller-related operations; Section IV-B tests the delay needed to collect answers to Zenoh's distributed queries; finally, Section IV-C compares Zenoh's performance against a HTTP REST-based implementation of the controllers' east/westbound communications.

A. Delay of controller-related operations

With reference to the topology in Fig. 1, we now focus on how the coexistence of Zenoh and of the SDN controller leads to different types of delay, depending on the zone where communication endpoints are located. We consider three classes of delays: i) *Host discovery delay*: required to retrieve information about a host (source, destination or both); ii) *Border discovery delay*: required to collect available paths to reach a certain zone (the source's, the destination's or both); and iii) *Controller forwarding delay*: required to compute an end-to-end communication path and forward data flow routing instructions to the requesting switch.

The host discovery delay is defined as the time difference between the start epoch of a distributed query and the epoch all responses arrive. Measuring the border discovery delay requires to complete a recursive discovery procedure starting from the destination, since the only information controllers have about other zones is the name of the zones they are adjacent to. To simplify the measurement of the border discovery delay, we carry out border discovery exactly once per experiment. Finally, the controller forwarding delay is defined as the time the controller takes to analyze a packet, compute the forwarding path (excluding the time required for queries) and transmit the packet.

We distinguish among the four communication types enumerated as follows, which require different actions:

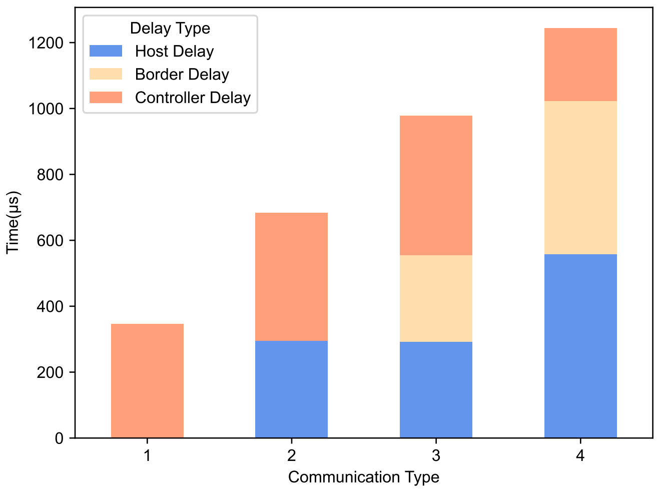

Fig. 3. Packet handling delays for different communication types.

1) both the source and the destination are in the same zone managed by the controller (no query needed);
2) the destination is in the zone of the controller, the source is not (one query needed for host discovery);
3) The source is in the same zone of the controller, the destination is not (one query needed for host discovery, one query for border discovery);
4) Both the source and the destination are in zones different from the one of the controller (two queries needed for host discovery, two queries for border discovery).

To measure the above delays, we need Zenoh and Ryu to co-exist. Because Ryu is written in python, we employ Zenoh-python, a binding to the core Rust implementation of Zenoh. It is worth highlighting that Python, being an interpreted language, cannot deliver the same performance as Rust, a compiled language. Therefore, the results in the following are to be considered as worst-case results imposed by the Python language, rather than by Zenoh.

Fig. 3 shows the average value of the host, border, and controller delays for the four above communication types. The controller delay is almost constant whenever the controller zone includes one or both communication endpoints. Our measurements indicate that the overall delay is bounded by the number of distributed queries performed by Zenoh. In fact, each controller is isolated, hence delays caused by changes in other zones do not affect local controller delays. Because the above results depend only on Zenoh's distributed queries, we will now turn to evaluate Zenoh's performance in more detail.

As a final remark, the above tests were performed by disabling the flow saving system, so that we can observe the longest delays, typical of an SDN's startup period. Once controllers learn how to route packets across zones, this knowledge can be cached,[1] so that no periodic queries to

[1]Recall that we assume that controllers and zones do not change location. Rather, only devices can move.

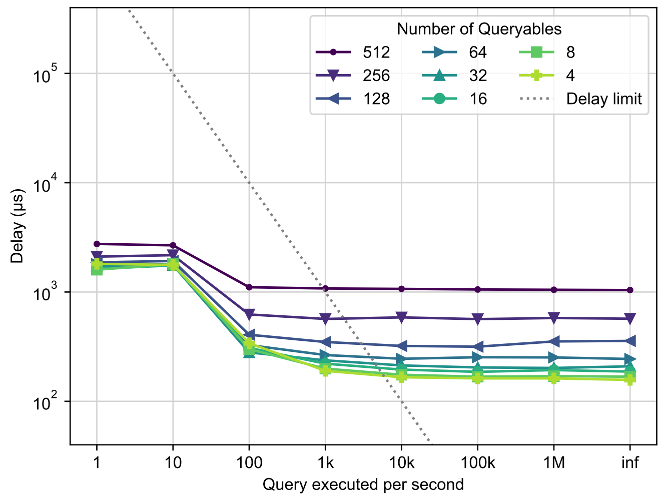

Fig. 4. Zenoh brokered topology, lazy consolidation. Best-case delay.

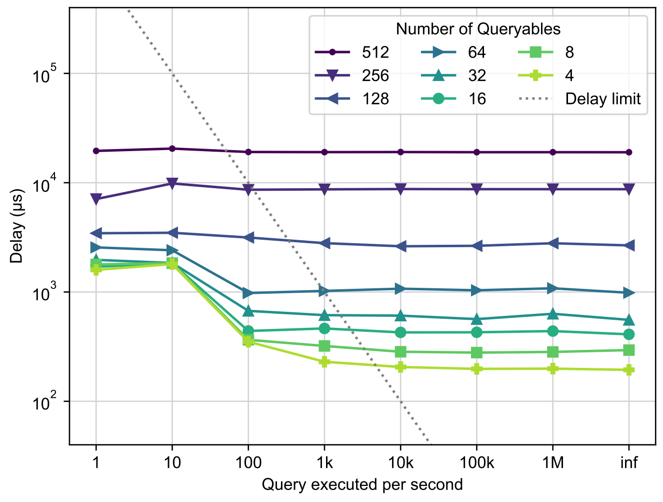

Fig. 5. Zenoh brokered topology, lazy consolidation. Worst-case delay.

retrieve end-to-end paths are needed, and the border delay disappears from inter-zone communication delays after startup.

B. Response delay for Zenoh's distributed queries

We consider a best case and a worst case for query response retrieval while applying lazy consolidation. In the best case, the queried resource is available at a single storage, hence only one controller will reply to a distributed query at any given time and the first returned sample is guaranteed to be the most recent one. This behavior is the same as with the None consolidation policy and retaining only the first sample. In the worst case, the same resource is available at all storages. Therefore, from the point of view of the response delay, the otherwise effective lazy consolidation policy behaves the same as full consolidation, i.e., it requires the querying controller to wait for all query responses.

We implement the above tests by starting one Getter (i.e., an SDN controller performing a query) and an increasing number of Queryables (i.e., SDN controllers replying to the query), in order to observe how the latter quantity affects the delay of the received responses. The payload for each resource in these tests is set to 32 bits.

Figs. 4 and 5 show the variation of the median delay as a function of the number of queries issued per second (i.e., the query rate, in the abscissa) and of the number of Queryables (different curves). The dotted line refers to the maximum delay affordable to deliver all responses before a new query is issued. Therefore, the abscissa of the intersection between each solid line and the dotted line conveys the maximum theoretical throughput of the corresponding test.

We observe that, for a fixed query rate, the response delay increases with an increasing number of Queryables (solid colored lines). Consider the best case in Fig. 4: as the query rate increases, the response delay decreases initially (up to 100–1000 queries per second) and then stabilizes. We attribute the initial decrease to the context switching overhead in the machines hosting the Getter and Queryables: as the query rate increases, it becomes more likely that the operating system keeps the Zenoh process in memory and in the CPU caches, thereby increasing its reactiveness to incoming queries.

Conversely, in the worst case of Fig. 5, the delay due to the multiple incoming responses dominates the overall delay if at least 128 Queryables are instantiated, and we observe the initial delay decrease only for 64 Queryables or less. We also observe that, with less than 64 Queryables, the best- and worst-case delays are almost the same, whereas tests with a higher number of Queryables experience one order of magnitude greater delay. In any event, the delay obtained for the largest number of 512 Queryables in the worst case is 10 to 20 ms, well below the 30 ms target dedicated to signaling for path instantiation and restoration [18], [19]. Considering that we can expect the heaviest east/westbound traffic only at controller startup, we can conclude that Zenoh is a suitable and scalable solution for the controller's east/westbound communications.

We proceed with Fig. 6, which depicts the cumulative distribution function (CDF) of the query response delays for an increasing number of Queryables, and considering a fixed query rate of 10k queries/s. In this case, every test runs in saturation conditions, above the delay limit shown in Figs. 4 and 5. All curves are very steep, signifying a very limited statistical dispersion of the query response times, that does not change even with increasing Queryables. Only for the largest number of Queryables do delay outliers slightly extend the tail of the CDFs. We can thus conclude that Zenoh is a suitable east/westbound protocol for use cases where response delay consistency and predictability are important design constraints.

C. Comparison against HTTP-based REST east/west APIs

We now compare the performance of Zenoh against a strongly coupled east/westbound communication alternative. For this, we implement HTTP-based REST APIs, which would require the controllers to maintain a one-to-one connection to

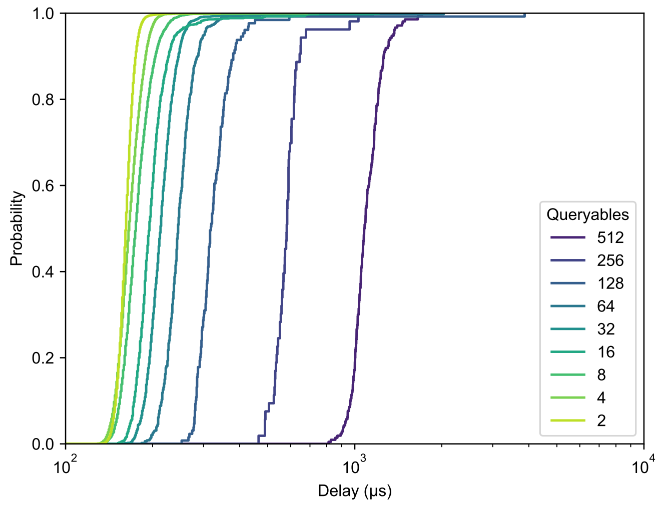

Fig. 6. Query delay CDF for an increasing number of Queryables. Zenoh brokered topology, lazy consolidation, best case.

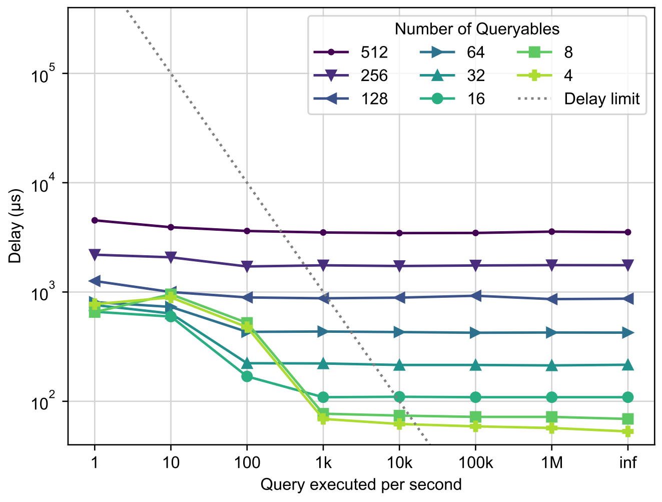

Fig. 7. East/westbound communications via REST APIs mediated by HTTP: query response delay.

every other controller. We tested this setup under the same conditions used for Zenoh. The HTTP library is written in Rust, setting the best scenario for HTTP, performance-wise. The results in Fig. 7 show that HTTP is typically slower than Zenoh's best case (cf. Fig. 4), except for the cases with up to 16 Queryables. While HTTP is indeed twice as fast as Zenoh's worst case (cf. Fig. 5), we recall that the latter is constructed by having all Queryable controllers reply with the desired resource and the same timestamp, so that no consolidation is possible. Besides this being an arguably unlikely occurrence, we also emphasize the following considerations.

Coupling—HTTP needs controllers to connect with each other into a fully meshed network, whereas Zenoh can work with any possible topology. In our test scenario, Zenoh has been configured in a brokered mode introducing one extra hop in the east/westbound path compared to HTTP. Nevertheless, Zenoh latency is still largely under the 30 ms delay target for 512 SDN controllers, which is a very unrealistic scenario. On the other hand, scaling a set of controllers that use HTTP would become increasingly more complex with an increasing number of controllers requiring heavy configuration and manual tracking.

APIs—Zenoh enables easy distributed query management via wildcards and pub/sub-based message passing, which are not natively available when using a strongly coupled HTTP REST interfaces. This allows any SDN controller application to be developed in a generic and flexible way that is independent of the number, locations, and roles of the SDN controllers.

Storage—Zenoh natively integrates support for geodistributed storage that provides data persistency to SDN controller. An HTTP REST interface requires manual work to discover, interconnect, and handle data replicas.

Consolidation—Zenoh can filter out responses with older timestamps to prevent network saturation. To achieve the the same behavior with an HTTP REST interface, significant burden needs to be put on the SDN controller.

Performance—Zenoh Python binding still outperforms the HTTP Rust implementation for many scenarios. As a consequence, should an SDN controller be available in Rust, Zenoh could deliver even better performance for east/westbound communication [20].

V. Conclusions

In this paper, we targeted software-defined networks (SDNs) with distributed controllers that can be connected using any topology, and evaluated Zenoh as a suitable east/westbound protocol to coordinate controllers, find end-to-end communication paths, and query/retrieve remote resources.

Our tests involve a topology with multiple controllers, an increasing number of Queryable controllers exposing resources within the SDN, and an increasing query rate targeting such resources. The results show that Zenoh achieves predictable query response delays with low statistical dispersion, even with a large number of Queryables. Further advantages of this solution include easy scalability to integrate additional controllers and support for any controller network topology. The main features that enable these advantages are Zenoh's location-transparent approach, distributed queries, and the automatic consolidation of query results.

Future work includes deploying an L2 controller fully integrated with Zenoh without bindings or wrappers and considering even larger network use-cases.

Acknowledgment

This work received support from the Italian Ministry of Education's Departments of Excellence initiative (Law 232/2016) and by the EU's Horizon 2020 research and innovation programme under grant agreement no. 101017109 (DAEMON).

References

[1] Y. Liu, A. Hecker, R. Guerzoni, Z. Despotovic, and S. Beker, "On optimal hierarchical SDN," in *Proc. IEEE ICC*, 2015, pp. 5374–5379.

[2] A. Abdelaziz, A. T. Fong, A. Gani, U. Garba, S. Khan, A. Akhunzada, H. Talebian, and K.-K. R. Choo, "Distributed controller clustering in software defined networks," *PLOS ONE*, vol. 12, pp. 1–19, 04 2017.

[3] M. Kulkarni, M. Baddeley, and I. Haque, "Embedded vs. external controllers in software-defined IoT networks," in *Proc. IEEE NetSoft*, 2021, pp. 298–302.

[4] F. Bannour, S. Souihi, and A. Mellouk, "Distributed SDN control: Survey, taxonomy, and challenges," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 1, 2018.

[5] Z. Latif, K. Sharif, F. Li, M. M. Karim, S. Biswas, and Y. Wang, "A comprehensive survey of interface protocols for software defined networks," *J. of Netw. and Computer Appl.*, vol. 156, pp. 1–28, 2020.

[6] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: A distributed control platform for large-scale production networks," in *Proc. USENIX OSDI*, Oct. 2010.

[7] Y. Ganjali and A. Tootoonchian, "HyperFlow: A distributed control plane for OpenFlow," in *Proc. USENIX INM/WREN*, Apr. 2010.

[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 *Proc. ACM HotSDN*, 2014.

[9] S. Yeganeh *et al.*, "Kandoo: A framework for efficient and scalable offloading of control applications," in *Proc. ACM HotSDN*, 2012.

[10] Y. Fu, J. Bi, K. Gao, Z. Chen, J. Wu, and B. Hao, "Orion: A hybrid hierarchical control plane of software-defined networking for large-scale networks," in *Proc. IEEE ICNP*, 2014.

[11] Q. P. Van, D. Verchere, H. Tran-Quang, and D. Zeghlache, "Container-based microservices SDN control plane for open disaggregated optical networks," in *Proc. ICTON*, 2019, pp. 1–4.

[12] A. Hölscher, M. Asplund, and F. Boeira, "Evaluation of an SDN-based microservice architecture," in *Proc. IEEE NetSoft*, 2022, pp. 151–156.

[13] G. Baldoni, J. Loudet, L. Cominardi, A. Corsaro, and Y. He, "Facilitating distributed data-flow programming with Eclipse Zenoh: The ERDOS case," in *Proc. MobileServerless*, 2021.

[14] L. Cominardi, R. Andres, K. Hopkins, and F. Desbien, "From DevOps to EdgeOps: A vision for edge computing," in *Eclipse Foundation edge native WG white paper*, 2021.

[15] S. Burckhardt, "Principles of eventual consistency," *Foundations and Trends® in Programming Languages*, vol. 1, no. 1-2, pp. 1–150, 2014. [Online]. Available: http://dx.doi.org/10.1561/2500000011

[16] K. Kaur, J. Singh, and N. Ghumman, "Mininet as software defined networking testing platform," in *Int. Conf. on Commun., Comput. and Syst.*, Aug. 2014.

[17] R. Kubo, T. Fujita, Y. Agawa, and H. Suzuki, "Ryu SDN framework – Open-source SDN platform software," *NTT Technical Reviews*, vol. 12, no. 08, 2014.

[18] W. D. Grover, *Mesh-Based Survivable Networks: Options and Strategies for Optical, MPLS, SONET, and ATM Networking*. Prentice Hall, 2003.

[19] S. Yadav, "Do we still need 50 ms restoration for running telecom services?" 2021. [Online]. Available: https://tinyurl.com/mtxawd6e

[20] "Zenoh's core Rust implementation performance," 2021. [Online]. Available: https://zenoh.io/blog/ 2021-07-13-zenoh-performance-async/