ZeroSDN: A Highly Flexible and Modular Architecture for Full-range Distribution of Event-based Network Control

Thomas Kohler, Frank Dürr, and Kurt Rothermel

Abstract—**Recent years have seen an evolution of SDN control plane architectures, starting from simple monolithic controllers, over modular monolithic controllers, to distributed controllers. We observe, however, that today's distributed controllers still exhibit inflexibility with respect to the distribution of control logic. Therefore, we propose a novel architecture of a distributed SDN controller, providing maximum flexibility with respect to distribution and improved manageability.**

Our architecture splits control logic into lightweight control modules, called *controllets*, based on a *micro-kernel* approach, reducing common controllet functionality to a bare minimum and factoring out all higher-level functionality. Lightweight controllets also allow for pushing control logic onto switches and enable local processing of data plane events to minimize control latency and communication overhead while leveraging SDN's global view to maximize control decision quality. Controllets are interconnected through a message bus supporting the publish/subscribe communication paradigm with specific extensions for content-based message filtering. Publish/subscribe allows for complete decoupling of controllets to further facilitate control plane distribution. Furthermore, we identify crucial requirements for practical on-switch deployments, where we employ lightweight virtualization techniques to ensure a safe control plane operation. We evaluate both, the scalability and performance properties of our architecture, including its deployment on a white-box networking hardware switch.

Index Terms—**Software-defined Networking; OpenFlow; Control Plane Distribution; Publish/Subscribe; White-box Networking; Virtualization**

I. Introduction

SOFTWARE-DEFINED NETWORKING (SDN) is based on the paradigm of *logically centralized control* of network elements. Logical centralization translates to the concept of *distribution transparency*, which is well-known from Distributed Systems. Distribution transparency hides the complexity of a physically distributed system from the application by making distribution aspects "transparent", i.e., *not* visible to the application. Thus, the client can be implemented as if the system were centralized. In particular, network control applications implementing network control logic have a global view of the network, although network information such as topology information inherently has to be acquired through monitoring by distributed network elements (the switches). Moreover, the SDN controller itself might be (ideally) a distributed system with all its defining properties like replication transparency, fragmentation transparency, and without a single point of failure. For instance, topology information stored in a "network information base" might be replicated to and partitioned between many servers to ensure availability and scalability.

Manuscript received March 28, 2018; revised August 9, 2018; accepted September 28, 2018. Date of publication October XX, 2018; date of current version October XX, 2018. The associate editor coordinating the review of this paper and approving it for publication was S. Schmid.

The authors are with the Institute of Parallel and Distributed Systems (IPVS), University of Stuttgart, Universitätsstraße 38, 70569 Stuttgart, Germany (e-mail: {firstname.lastname}@ipvs.uni-stuttgart.de).

Digital Object Identifier 10.1109/TNSM.2018.XXXXXXX

A. Evolution of SDN Controller Architectures

Many SDN controllers have been implemented so far based on the concept of logically centralized control. Figure 1 depicts the evolution of controller architectures with respect to distribution and modularization.

First SDN controllers were *monolithic systems*, implementing the controller as one process. The SDN controller connects through the southbound interface to the switches using, for instance, the popular OpenFlow protocol [1], and the control applications interface with the SDN controller through a northbound interface, e.g., a Java API or REST interface. To increase fault-tolerance, the monolithic process implementing all control logic can also be fully replicated.

Very similar to the evolution of monolithic operating system kernels like the Linux kernel, this monolithic design was soon extended to a *modular monolithic design* (Figure 1(a)), where control modules implementing certain control functions (*network functions*) can be dynamically loaded into the controller process at runtime. Two examples showing that this design is still used in practice are the popular ONOS and OpenDaylight controllers [2], [3] relying on OSGi [4]. However, their modular controller architectures remain monolithic since they still rely on a central controller executing all modular control functions in one process. Again, the logically centralized controller can be physically distributed with each replica containing all control functions, i.e., replicas are identical clones.

Mainly to further increase scalability and reliability, SDN controller evolution continued to investigate *distributed SDN controllers* (Figure 1(b)). Network control can be distributed along two dimensions. First, similar to the modular monolithic design, individual control functions can be factored out into control modules, which are now partitioned between different physical machines instead of fully replicating all control functions on all machines. Secondly, control can be partitioned over the network topology, i.e., the scope of individual control modules can be limited to disjoint subsets of switches. While facilitating scalability with the network size, partitioning raises the need for coordinating the scope of control.

B. Motivation: A Full-range Distribution Architecture for SDN Controllers

Observing that distributed SDN controllers already exist today, can we conclude that their evolution has reached its end? We argue that this is not the case, for the following reasons.

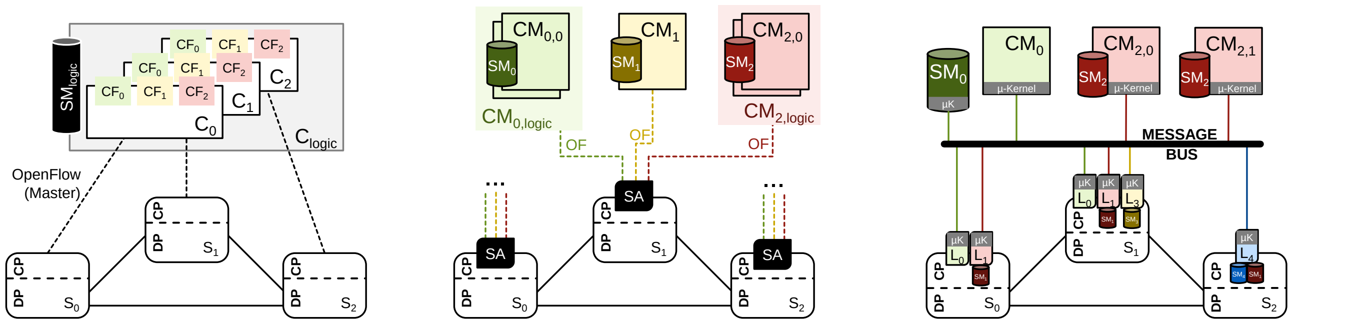
First of all, implementing *fully distributed network control* (without switch-external control functions) is not anticipated. In other words, the traditional SDN approach mandates an external network controller (monolithic or distributed). Direct switch-to-switch communication for network control is not possible. This reflects the clean-slate paradigm shift from distributed network control to logically centralized control, where switches are just "dumb" network elements, specialized to do fast forwarding, according to rules defined by an "intelligent" remote controller implementing all network control logic. On the one hand, this reduces the functionality of switches to a bare minimum, allowing for minimal switch resources and design. On the other hand, outsourcing all control from the switch comes at the cost of increasing latency due to incurring switch-controller round-trip times (slower reaction), increasing load on the control network, or difficult implementation of robust logically centralized control relying on additional machines that can fail. Therefore, we argue that a highly flexible SDN architecture would allow for the full spectrum of distribution, from fully centralized to fully distributed control. In other words, control logic has to be brought back onto the switch. Although execution of control logic on the switch hardware on the one hand has been conceptually proposed in literature [5]–[7], due to lack of distribution support or high computational resource demand, in concrete implementations it has been reduced to offloading of certain functionality, such as packet generation [8] or state machine logic [9]. To fully exploit the locality of switches, we argue to include the switch in the control distribution and allow for decision making on the local scope. Besides the extremes—fully (de-)centralized control—we argue that network control decisions are ideally be taken as local as possible, in order to *minimize control latency*, while leveraging the logically centralized paradigm of SDN through access to global knowledge in order to *improve decision quality*. Since requirements, such as timeliness, optimality, and consistency, may differ between network functions, a network control architecture should provide the flexibility for balancing these trade-offs for each individual network function. For instance, for forwarding decisions at a switch, full global knowledge is typically not required. The focus rather lays on timeliness in order to reduce forwarding latency. In contrast, traffic engineering or monitoring are applied on a much broader time scale and thus looser latency constraints, but relying on more global knowledge for improved solution quality.

Secondly, with the current concept we observe that controllers tend to be quite heavyweight (which might also be a practical reason why control has been removed from switches). For instance, in order to just receive packet-in events, the ONOS controller requires a full-fledged OSGi environment with a total code size of $\approx 216\,$MByte. Controllers that are more lightweight typically lack modularity or distribution capability. We argue that it should be possible to identify a minimal feature set that every control module can implement to communicate with switches and other distributed control modules. Anything else should be factored out into the implementation of the control function. In other words, we advocate a lightweight *micro-kernel* approach for SDN controllers instead of a heavyweight monolithic controller architecture.

Thirdly, we observe that switches and controllers are still tightly coupled, which hinders the free distribution of control logic. For instance, an OpenFlow control channel requires a TCP connection to a controller. Since TCP is inherently based on connections to certain machines, spawning new control applications at other machines or migrating them between machines is cumbersome and potentially disruptive [10], [11]. We argue that switches must be *decoupled* from the SDN controller. This can be achieved by using state-of-the-art communication middleware approaches as already successfully used in other domains for the communication between services [12]. As a side effect, choosing a suitable communication middleware also allows for implementing control logic in virtually any language and to support event-driven as well as request/response types of interaction.

This article is an extended version of previous publications [13], [14] and provides detailed insights into various aspects. Its **main contribution** is a novel architecture for a distributed SDN controller fulfilling all of the above requirements: (1) high flexibility with respect to distribution of control logic covering the whole design space from logically centralized to fully distributed control; (2) micro-kernel controller architecture for distributed lightweight controller modules (so-called controllets); (3) push-down of controllets implementing control logic onto switches, allowing for fast local decision making while leveraging global knowledge; (4) decoupling of controllets through a message bus supporting content-based filtering of so-called data plane events. Furthermore, we address challenges in practical deployments of switch-local controllets, where we employ lightweight virtualization techniques to cope with hardware heterogeneity and to implement isolation and resource control for a safe and controlled control plane operation. An implementation of the proposed concepts is publicly available on GitHub (https://zerosdn.github.io/) [15].

The rest of the article is structured as follows. In §II, we describe the architecture of our distributed SDN controller together with an overview of the basic concepts. We proceed with describing the message bus in more detail in §III. In §IV, we discuss how our concept enables highest flexibility in terms of control distribution, before we present local logic based on global knowledge, along with multiple applications. We also address the relation to Data Plane Programming and Network Function Virtualization as well as challenges in practical deployment. In §V we elaborate on implementation aspects, followed by an evaluation of performance and scalability of our distributed architecture as well as results from the deployment of on white-box networking switch hardware in §VI. We discuss related work in §VII and conclude the article in §VIII.



(a) Modular, monolithic, replicated; *C: controller instance, CF: control function, SM: state module; S_i: switch*

(b) Modular, distributed (partitioned, replicated); *CM: control module, SA: local OpenFlow-wrapper (switch adapter)*

(c) μ-kernel architecture with fully distributed local *(L)* & external controllets *(CM)*, interconnected by a message bus.

Fig. 1. Evolution of distribution in SDN controller architectures. Rightmost: ZeroSDN's full-range distributed architecture.

II. Architecture

We start by introducing the basic architecture of our distributed SDN controller (see Fig. 1(c)).

Our approach is based on what we call a *micro-kernel architecture* for SDN controllers. We split network control logic into lightweight control modules, whose instances we call *controllets* (*CM_i*). In contrast to a monolithic controller, controllets do not require a heavyweight execution environment. Instead, we execute each controllet in a separate process, possibly being also physically distributed, and enable communication between them. The micro-kernel (μK) just provides basic functions for messaging including publish/subscribe message routing and parsing (in particular of OpenFlow messages), and registration and discovery of controllets and switches. Any other functionality like network topology management, routing, etc. is implemented by the controllets' "business" logic. One advantage of having a slim functionality for the SDN micro-kernel is that we can port the micro-kernel with little effort to different languages enabling us to basically use any language for the implementation of controllets. Moreover, the lightweight nature of controllets enables us to push down control logic by executing controllets directly on switches (S_i), instead on remote server hardware. We denote controllets running locally on switches as L_i. Opposed to a monolithic solution, the distribution of control logic comes at the cost of increased complexity for the distribution of its state. We discuss trade-offs in control distribution in Section IV-A.

Communication is based on a unified *message bus* to decouple controllets from switches and other controllets, both, logically and physically. We are thus able to reduce the switch-controller coupling to inter-module communication over the message bus. Each controllet and switch can communicate with other controllets or switches through the message bus by sending events using the publish/subscribe (pub/sub) paradigm, or sending direct messages using the request/response paradigm. Decoupling controllets and switches allows for flexible distribution including migration of controllets, and dynamic spawning or exchanging of controllets at runtime. The message bus implementation is integrated into the micro-kernel.

Overall, this architecture allows for maximum flexibility. Next, we refine our architecture and elaborate on the technical details and further key features enabled by our approach.

III. The SDN Message Bus: Decoupling Controllets through Events

Our architecture is based on *event-based communication* to decouple the producers of events from their consumers in both, time (asynchronous communication) and space (distribution of logic between nodes). In the domain of SDN, we particularly consider so-called *data plane events* (DPE) stemming from packets or state changes of data plane elements (switches and end systems). They include the addition or removal of network elements, link status updates, and packet ingress or egress. From certain DPEs state information can be inferred, such as knowledge of the physical network topology and end system protocol state, e.g., TCP-sessions. A DPE is either processed in the hardware forwarding-pipeline of the switch, e.g., a packet ingress is processed according to the flow rules installed in the switch's TCAM (*fast-path*), or is being forwarded to the control plane (*slow-path*), e.g., when no matching forwarding rule exists. In the latter case, the switch silicon passes the associated packet to the switch's CPU, where it is encapsulated into an OpenFlow `PACKET_IN` message. When not processed locally (see §IV-B), the switch publishes the DPE to the message bus, which delivers it to controllets that are subscribed to this kind of event. The message bus is responsible for routing event notifications to their subscribers. Since DPEs often include matches on packet header fields, we argue that the message bus should support *content-based filtering* of events [16]. Therefore, event conditions include matches on header field tuples or any other meta-data. This paradigm can also emulate standard client/server communication (request/response), multicast, or topics [16] using filters on receivers, groups, topics, etc.

Event routing in the message bus is exemplarily illustrated in Figure 2: an ingress TCP segment from an end system at S_0 is encapsulated in a DPE (`OF_PKT_IN`) and published to the message bus, where a remote monitoring firewall controllet (Mon) and one instance of a remote forwarding controllet (Forw2) have matching subscriptions, i.e., are responsible for such events, and are consequently delivered the event. As a result of processing this event, Forw2 sends a packet-out message (`OF_PKT_OUT`) over the message bus directly to S_2 using the request/response pattern. Analogously, Forw2 installs a flow from S_0 to S_2 by sending flow modification messages (`OF_FLOW_MOD`; omitted in Figure 2 for readability).

However, we do not restrict ourselves to *basic data plane events*, but also consider *complex data plane events* involving, for instance, multiple packets and timing conditions. For instance, a complex event could be triggered by a certain sequence of packets, or the non-arrival, i.e., absence, of a certain packet over a defined period of time, also across multiple switches. Typically, switches only fire basic events, which are then forwarded to subscribing controllets, which in turn evaluate complex event conditions to fire complex data plane events. Due to space constraints, we do not further elaborate on complex data plane events in this article.

Another type of events, used for inter-controllet communication, is the control plane event (CPE), which bears state changes or other events of the controllets' business logic or their micro-kernel, such as topology changes, firewall policy changes, or recovery/shutdown of controllets. CPEs are mainly used for coordination among controllets. In our example, a link failure DPE (`OF_PORT_STATUS`) at S_1 is disseminated to the message bus and delivered to the subscribed Topo controllet, which hence adapts its knowledge about the network topology. Consequently, Topo informs interested controllets by publishing a CPE (`TOPO_CHANGED`), for which all forwarding controllets have subscribed to react to topology changes, and so on.

Recent SDN research has shown that consistency in an inherently distributed system of switches and controllers might require certain semantics of the delivery of messages [17]. The message bus transparently implements a range of semantics, such as *exactly once* or the relaxed *at most once*, by employing corresponding messaging primitives, such as atomic multicast, (un-)ordered multicast, etc. Thus, the message bus provides arbitrary guarantees on message delivery (reliability) to controllets as building blocks for implementing network control with flexible consistency semantics that match the criticality of respective control tasks.

Since the message bus is a crucial system component, we want to briefly discuss its implications regarding scalability and reliability. In traditional messaging middleware, publish/subscribe used to be implemented by a hardware appliance or a software-based component, the *broker*, which manages subscriptions and implements filtering of messages in a centralized fashion. To prevent swapping one centralized component (the centralized SDN controller) for another (the centralized message broker), we employ a distributed solution that exhibits high scalability: Modern *brokerless* message bus implementations use efficient transport mechanisms for event dissemination, like multicast or unicast with publisher-side subscription based filtering or even hardware-based filtering with line-rate performance [18], targeting scalability to hundred thousands of subscriptions [19], which suffices to accommodate typical data center networks [20]. We provide macro-evaluations of our message bus implementation in Section VI. Should performance issues arise nonetheless, e.g., due to an insufficiently dimensioned control network, scalability can be improved by employing a message bus hierarchy, where the scope of controllets is limited, e.g., reflecting tiers on modern data center network topologies, such as core, spine, and leaves. Regarding failure tolerance, we stress that failure of the message bus translates to a broken control channel, which is equally

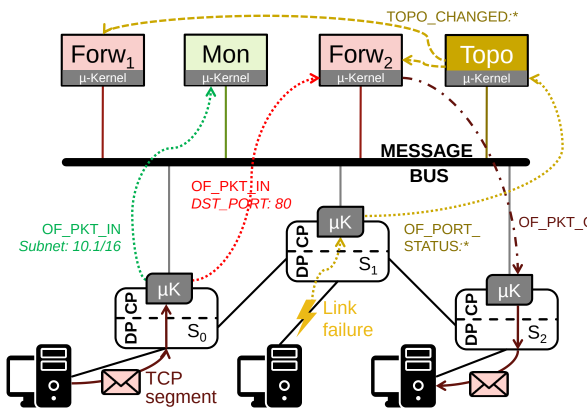


Fig. 2. Content-based routing (publish/subscribe) of data plane events (dotted) and control plane events (dashed) for exemplary subscriptions and direct messaging (request/response; dash-dotted) over the message bus, decoupling remote controllets.

severe as a broken control channel in traditional, less distributed SDN architectures. On the contrary though, local control in our architecture increases failure-tolerance, as we show later.

IV. Highly Flexible Control Plane Distribution

In this section, we make a solid case for rethinking the radical clean-slate approach most common SDN architectures follow by showing how lightweight controllets can bring back control onto the switch while still benefiting from the logically centralized paradigm of SDN. We also address drawbacks of control decentralization and challenges in practical deployments.

A. Augmented Fully Distributed Control

Most SDN architectures have abandoned fully decentralized network control based on a distributed control plane implemented solely by switches in favor of logically centralized control. While not strictly arguing for or against logically centralized control or fully distributed control, we observe that the strict notion of separating data plane elements from the logically centralized control plane limits the full potential of the SDN paradigm. For instance, "legacy" distributed control protocols, such as distance vector or link state routing protocols, have proven to be fault-tolerant and scalable. As investigated by [17], vigorous efforts have to be undertaken to provide the same fault-tolerance with a logically centralized SDN network. We stress the fact that maintaining a global view and exerting logically centralized control comes at a cost [21] due to the inherent need for acquiring a global state, which gets costlier the stricter the consistency requirements are, and communication with a remote control entity, respectively. A full global view is however not even needed for many control decisions, as we show later. Hence, logically centralized control should not be the sole option. Consequently, we argue that true flexibility in network control implies to leverage the whole design space of control (de-)centralization and thus also includes the option for full distribution of network control, as depicted in Figure 3.

Recent developments in networking hardware enable switch-local control logic due to a) increased computing performance and b) programmability through open access to the switch's control plane. In particular, *white-box networking switches* feature open, Linux-based switch operating systems as the control plane, running on increasingly powerful CPUs (see §VI-C). Therefore, and in-line with recent research [8], [9], [22], [23], our architecture encourages pushing lightweight controllets directly onto the switch, as illustrated in Figure 1(c). These switch-local controllets can then execute the full spectrum from simple local logic to fully distributed network control protocols. Like any controllet, also switch-local controllets communicate through the message bus. Thus, we can implement distributed network control alongside logically centralized network control, or implement anything in-between (Figure 3, light-shaded area). This scheme allows for the best of both worlds—fully decentralized processing, yet being centrally coordinated, and logical centralization, which allows for trading-off *control latency* (latency of event processing and communication) against overheads of distribution and synchronization of controller state. For the synchronization of state among controllets, communication primitives of varying reliability offered by the message bus can be combined with additional methods to achieve a desired level of consistency and other properties, for instance by employing a 2-phase commit protocol for distributed transactions [24]. The selection of a suitable level of synchronicity (*synchronization requirement*; Figure 3, dark-shaded area) depends on the criticality of a network function to control. For instance, network operators could consider admission control more critical than monitoring or traffic engineering, where temporal inconsistencies are bearable, i.e., changes in these policies do not have to be enacted as quickly (eventual consistency).

Besides the partitioning of controller state data along network functions, state can additionally be partitioned by topological scope. Through incorporation of (more) global knowledge, i.e., state data of larger topological scope, we can thus additionally trade-off the *scope of state data* against *solution quality* of control decisions (Figure 3, dark-shaded area). As we show next, local knowledge can be augmented by partial caching or aggregation of (more) global knowledge upfront or by requesting remotely within a control decision process.

Potential control decision conflicts can be resolved by publishing all policy information and aggregating them locally alike. Local controllets decide which policy information is relevant for their control decisions, issue corresponding subscriptions, and cache received policy data.

The flexibility of our approach is to the best of our knowledge yet unmet and exploits the full conceptual range of SDN.

B. Local Data Plane Event Processing

We argue for placing control decision making as close as possible to the entities it is affecting, i.e., pushing down *decision making* instead of *decisions* (in form of forwarding entries) to the switches. We denote this concept as local data plane event processing (LDPEP). LDPEP allows for reacting most timely on data plane events, decreasing control latency. Another important advantage is that the state data of local scope naturally is most recent locally and constitutes the ground truth for decision making. Due to its locality, it neither has to be costly acquired

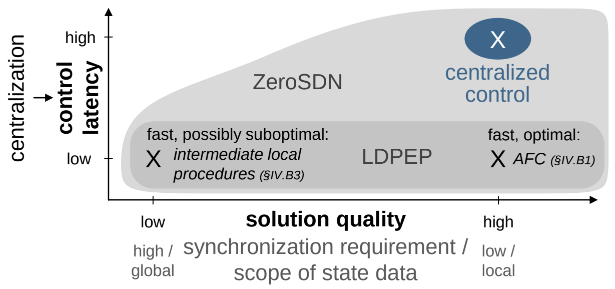


Fig. 3. Design space and trade-offs in network control distribution. Opposed to logically centralized control with high control latency (dark blue), ZeroSDN covers the full spectrum (light-shaded area) and offers an additional degree of freedom by exploiting the trade-offs of state data scope and synchronization requirements against solution quality (dark-shaded area).

nor has it to be consistently agreed upon. Furthermore, opposed to a non-local controllet, the total control load is inherently balanced to local controllets, relieving the message bus.

We apply a fast heuristic to quickly decide whether an event is to be processed locally or remotely. Therefore, we consider the scope of the state data required for decision making, as well as the scope of the particular control decision. If the involved state data and decision are of limited scope and all necessary state data is locally available, the event is processed locally. Otherwise, the event is propagated over the message bus to be processed by remote entities in the control plane. Note that this decision is not exclusive and also the control scope is not necessarily limited to a single switch. Even with LDPEP, we still allow controllets to have forwarding rules being installed directly at the switch.

LDPEP not only decreases latency but also increases the network's failure resilience: it constitutes a stand-alone procedure in case an adequate remote controllet or the entire message bus is currently unavailable.

In the following, we will show essential use cases enabled by LDPEP and elaborate on its design by example.

1) Autonomous Forwarding: A prime candidate that naturally lends itself to LDPEP is simple forwarding as, e.g., being implemented by the *MAC learning switch Nicira extension* [25] in the prominent SDN software switch implementation Open vSwitch (OVS) [26].

In the following, we will present the concept of *Autonomous Forwarding*, which is illustrated in Figure 4, running on a typical switch hardware platform. Following standard OpenFlow behavior, packets (❶ from $Host_{src}$ destined to $Host_{dst}$) without matching forwarding rules in the fast-path ❷ are escalated over the slow-path to the switch's control plane (❸ `PACKET_IN`), where a forwarding decision is taken and applied by installing respective forwarding rules (❼ `FLOW_MOD`) for subsequent packets and sending the particular packet to a switch data plane egress port (❽ `PACKET_OUT`). Naively one could conclude the only state information needed for the forwarding decision was the end host MAC to switch-port mapping, which is either passively learned ❹ from ingress packets or actively probed. However, the destination host might not be attached to a port of that switch. In addition, forwarding decisions might violate global network policies, such as firewall rules, ACLs, or tenant isolation.

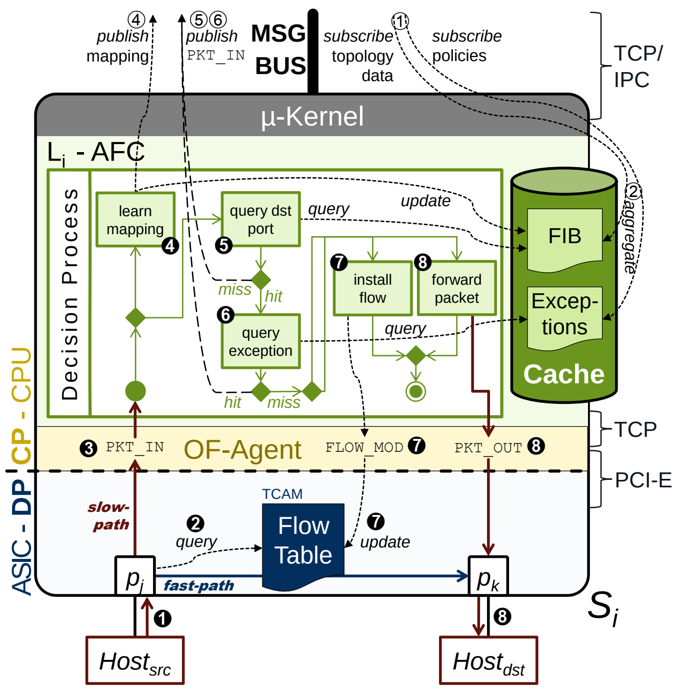


Fig. 4. Schematic overview of the Autonomous Forwarding controllet and its processing of a local data plane event (forwarding of $Host_{src} \rightarrow Host_{dst}$) on a typical switch hardware platform.

To implement centrally coordinated control, preventing policy conflicts, and leverage global network view, the Autonomous Forwarding controllet (AFC) subscribes to relevant topology data and policy information on the message bus ①. Due to limited resources on the switch, the extent of local state caching has to be limited. Received publications about possibly interfering policies are thus aggregated ② into an *exception list*, storing end hosts and local switch-ports that are affected by any policy and are thus being blacklisted (or whitelisted). Similarly, topology information is reduced to only relevant parts for local processing before being stored in the cache.

In the forwarding decision process, the MAC-switch-port mapping of $Host_{src}$ is learned and the Forward Information Base (FIB) cache is updated ❹. Note that FIB entries (tuples) may be arbitrarily extended, for instance to consider VLAN tags. Since the mapping constitutes topology information that in general is highly relevant for many other controllets as well, it is published to the message bus ④. Then, the cached topology data, i.e. the FIB, is queried for the switch-port associated with $Host_{dst}$ ❺. In case the data is not present locally, the PACKET_IN event can be escalated to the message bus to be processed by some remote controllet ⑤ or a request for the required data can be published. To evaluate whether autonomous local processing can be applied, the *exception list* is queried ❻. In case of a hit, the decision must not be taken locally and is thus escalated to the external control plane by a publication of the event to the message bus ⑥. Otherwise, local processing proceeds ❼❽. Note that for policies that can be translated directly into local drop-rules, such as admission control, affected DPEs with corresponding matches in the exception list can still be processed entirely locally.

While maintaining a local exception list is mandatory for policy adherence, the scope of non-local topology information to be locally cached can be chosen more fine-grainedly, considering the available resources on the particular switch and the desired data consistency. The scope of the local topology cache thus can range from purely local over regional (neighbor switches) to global view. This allows for trading off optimality of a control decision against resource consumption (memory, processing) and latency (for decision making and enacting). As mentioned above, data consistency is a crucial factor for the optimality and even validity of a decision. Typical cache invalidation and eviction strategies such as *least recently used* or *least frequently used* can be applied to optimize caching behavior. As a middle ground, instead of topology data itself, the cache can just store the primary source for that data—the controllet at which the data is local. Thus, in case such data is needed, the respective peer can be queried directly rather than publishing an uninformed query to the whole message bus.

2) ARP Handling: ARP is another essential networking mechanism, which has already been investigated in the context of local control and controller-function offloading [8], [27]. Autonomous forwarding can be easily extended to include ARP handling. Additional to the link layer address data, ARP needs network protocol address data, which is passively or actively acquired, alike. Since ARP is a control protocol, we argue to employ a reactive control scheme, where all ARP requests are escalated to and handled in the control plane. Thus, at the cost of negligible memory consumption, ARP handling profits from decreased latency of LDPEP, while the remote controllets are effectively shielded from ARP control load that, in contrast to proactive flows, is to be fully handled by the control plane. Extensive evaluations of quantitative impact of local ARP handling can be found in the mentioned literature.

3) Fast Failover & Adaptive Link Load Balancing: While decisions of the AFC and ARP LDPEPs are **permanent**, i.e., typically not challenged by external authorities (remote controllets), we now describe another class of LDPEP: **intermediate local procedures**. These allow for fast local reaction, while possibly compute-intensive and thus time-intensive centralized control decision is eventually determined and possibly replacing the local short-term procedure decision.

In our exemplary **local fast failover** procedure, a link failure (yet another type of data plane event) between a pair of adjacent switches (S_1, S_2) is detected at S_1 and propagated to a controllet running on S_1. A local procedure temporarily compensates the failure by steering the traffic over a link locally known[1] to belong to a redundant path to S_2. S_2 recovers analogously. Although being possibly suboptimal, local intermediate procedures provide a timely recovery, while the failure event is propagated to the message bus, where a remote controllet recalculates a globally optimal route that is ultimately deployed to the switches possibly overriding the decision of the local procedures. If S_1 and S_2 have broader cache scope, they could even avoid most suboptimal recoveries by coordinating their plans among each other using peer-to-

[1]Switch to switch links can be discovered by employing active probing using the Link Layer Discovery Protocol, as described in Section V.

peer communication, and adapt it in case of discovered sub-optimality. A related approach [28] relying on pre-installation of failover flows and thus consuming additional scarce flow table space shows that recovering through remote controllers is one order of magnitude slower than local procedures.

Instead of being applied to recover from (rare) failures, re-steering flows over redundant links according to the present link utilization can be a time-event-triggered (periodic) process, which we denote as **adaptive link load balancing**. This procedure is highly appealing for traffic engineering and more dynamic than traditional approaches, such as *Equal-cost Multipath Routing* (ECMP) [29]. Recent switch instrumentation technologies, like Broadcom's BroadView [30], even enable fine-grained access to hardware switch-port queue statistics, which allows for more detailed traffic analysis. Furthermore, adaptive link load balancing can be applied not only on local scopes, but rather on different levels of a whole control hierarchy, e.g., reflecting tiers on data center network topologies.

4) Control Plane Feedback Mechanism: Local controllets are the only entities that can *directly* access the switch's flow table entries. Thus, any applied change to a flow table can be propagated to interested controllets, implementing a feedback mechanism that allows a controllet to verify whether its flow change has been successfully applied—a precursor for a transactional interface [31]. Although policy conflicts between controllets should be avoided by coordination upfront, with this mechanism, controllets are able to detect conflicts, e.g., when a rule, encoding a policy of one controllet CM_1 is modified by another controllet CM_2 such that the original policy of CM_1 is violated.

C. Relation to Data Plane Programming and Network Function Virtualization

Data Plane Programming, like advocated by the popular P4 initiative [32], has become a huge trend in SDN. It features protocol-independent and flexible packet processing in networking hardware, opposing OpenFlow's matching mechanism which is limited to static headers of established network protocols and the rather static hardware processing-pipelines of traditional switch silicon. In a nutshell, Data Plane Programming leverages the increased capabilities and programmability of modern networking hardware, such as network processors, FPGA-augmented switch silicon, programmable switching ASICS like the popular Tofino ASIC, or programmable NICs to extend the expressiveness of packet processing in the data plane, i.e., on the fast-path. Data Plane Programming paves the way for complex yet efficient processing of high-volume data in the network at line-rate, for instance in the domain of data analytics or stream processing [33], thus recently coining the term In-Network Computation [34].

Hence, Data Plane Programming provides a particularly interesting opportunity for Network Function Virtualization (NFV), where network functions such as firewalls, NAT gateways, or load balancers, are flexibly moved from costly dedicated hardware middleboxes onto commodity server hardware using virtualization techniques. Even with standard OpenFlow, complex network-centric appliances such as content-based routing can be entirely substituted by on-route packet processing directly on switches data plane, providing line-rate throughput [18] and eliminating the need for a remote middlebox or a virtualized network function (the broker). Data Plane Programming shifts this frontier even further, enabling pushing down more complex network functions to switches.

Recent NFV-related SDN approaches typically focus on the distribution of network functions onto the switch data plane, like the generic frameworks OpenBox [35] and NetBricks [36], or the management and orchestration of virtualized network functions (vNFs), like the E2 [37] framework, which handles the dynamism, placement, and chaining of vNFs. ZeroSDN is mostly complementary to NFV. While it also supports the implementation of dynamic network functions in the switch data plane[2], for instance implementing a stateless firewall with the AFC, with LDPEP, ZeroSDN rather focuses on the distribution of control plane functions for fast adaption, as shown with the intermediate local fail-over procedure or local load balancing. Contrasting OpenFlow and P4, ZeroSDN thus incorporates local control decision making, rather than mere local deployment of remote control decisions.

D. Challenges of Deployment on Networking Hardware

1) Migration and Closed Switch Hardware: In order to be able to run controllets locally, the switch's control plane has to be accessible, which is a defining property of white-box switches. The proliferation of white-box switches is reflected in the increasing number of hardware and software specifications for white-box switches, that are provided to the public domain by big players like Facebook and Microsoft [38]. The white-box market share is expected to double within the next five years [39]. However, for switches with an inaccessible control plane or insufficient resources, we provide a fallback mechanism that enables integration in our architecture. Such a switch is coupled with a dedicated SwitchAdapter, which instead of running locally is running on any other hardware, preferably in close proximity to the switch, via an OpenFlow connection and acts as a gateway to the switch in the message bus. Note that an external SwitchAdapter is still capable of executing local logic, yet additional network latency is incurred. We determine the penalty of externalizing the SwitchAdapter in Section V-B.

2) Isolation and Resource Control through Lightweight Virtualization: The accessibility of the control plane is white-box switches' boon and bane: it allows arbitrary processes of different provenance to run in a less controlled environment, opposed to the closed switch model of traditional full-stack vendor implemented proprietary switch platforms. This raises concerns regarding security and reliability. (Unintentional) adverse behavior of control plane processes, including failures and excessive resource consumption, could starve other essential processes and thus poses a severe threat to its entire operation. Consequently, we derive two requirements for the practical deployment of LDPEP: 1) **Isolation** to protect processes' data from each other and ensure data integrity, and 2) **Prioritization**

[2]In principle, LDPEP can implement arbitrary packet processing—in the control plane. For a detailed discussion on LDPEP's generalization to arbitrary slow-path packet processing and the sweet spot of packet processing, we refer to related work of ours [14].

and resource control to ensure liveness of control operation and thus ultimately network operation. Furthermore, the current white-box switch landscape exhibits a high heterogeneity with respect to hardware, i.e. switch silicon and control plane architecture (x86, ARM, PowerPC), and software, i.e. operating systems and forwarding agents.

We combine local controllets with lightweight virtualization to cope with white-box networking heterogeneity and to achieve required isolation properties. Since the large overhead of virtualizing a full OS along with an application (traditional virtual machines) counteracts the latency gains of local logic, we focus on using two lightweight techniques: 1) *Library OS / Unikernel*, such as Rump Kernel [40], where the guest OS is stripped down to a bare minimum, i.e., providing just the functionality the virtualized application needs for its operation and 2) *Containers*, such as LXC and Docker, which abandon hardware emulation and full OS virtualization in favor of using isolation features of a shared kernel, providing multiple isolated user-space instances. They allow for fine-grained control over both, the scope of isolation (*namespaces*) and resources allocation (*cgroups*) at minimal overhead. We evaluate the overhead of these techniques in Section VI-C.

V. Implementation

We have implemented an open-source prototype of our distributed SDN controller architecture, consisting of a modular execution framework (ZMF) running a distributed SDN controller application (ZSDN) with essential controllets atop [15]. ZMF and most modules are written in C++, but we also provide a Java-based module framework (JMF). We provide build support for x86 and ARM architectures. This section presents the most important aspects of our implementation. Additional technical documentation is available online [15].

A. ZMF: The Zero Module Framework

Our micro-kernel implementation consists of two components, the `PeerDiscoveryService` and `MessagingService`. Module runtime environments are completely decoupled and independent of each other. They run in dedicated processes, possibly on separate hardware. The `PeerDiscoveryService` implements module discovery with dependency and life-cycle management, enabling bootstrapping and peer dynamics. To this end, changes in a module's lifecycle state, such as joining/leaving the framework, are propagated using efficient UDP multicast. Furthermore, modules periodically confirm their state by multicasting heartbeat messages. Thus, with linear message complexity, each module knows the type and state of all other modules.

For the message bus we employ the production-grade low-latency communication middleware *ZeroMQ* (ZMQ) [41]. Besides numerous communication patterns and transport mechanisms of varying reliability, ZeroMQ comes with a security framework implementing authentication, confidentiality, message integrity, etc. [42]. Access to the message bus is provided to ZMF modules through the `MessagingService`. We use TCP and IPC as reliable transport mechanisms. Later, we will show the mapping of data plane events and control plane events to pub/sub topics.

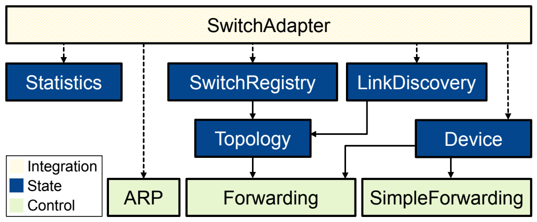


Fig. 5. Dependency graph for essential controllets.

B. ZSDN: A Distributed SDN Controller

ZSDN consists of prototypical controllets for distributed SDN control. All controllets support OpenFlow (OF) 1.0 and 1.3. Common data structures like topology data are mapped to *Google Protocol Buffers* [43] definitions, providing language-independent module communication.

Figure 5 shows essential controllets and their logical inter-dependencies. The SwitchAdapter (SA) wraps an OF-enabled switch in an instance which is running locally on the switch, integrating it to and representing it within the framework.

State controllets acquire data plane state by passively reacting on subscribed events or active probing. For instance, the SwitchRegistry registers all available switches through subscriptions on changes of their representing SwitchAdapters, whereas the LinkDiscovery controllet detects switch to switch links by subscribing to LLDP (Link-Layer Discovery Protocol) data plane events and proactively injecting LLDP packets over the SA instances into the data plane. The Topology controllet subscribes to both, SwitchRegistry and LinkDiscovery events, such that eventually it holds complete topology knowledge, excluding end systems, which are managed by the Device controllet. Topology information can be actively queried by controllets using req/rep. Topology changes are published through events, allowing for passive synchronization of controllet-local caches. Another module class provides control feedback to the data plane and thus closes the network control loop by modifying forwarding rules, such as the SimpleForwarding controllet.

1) Event Space – Topics Mapping: Due to the lack of practical high performance content-based pub/sub middleware implementations, we use ZMQ's topic-based pub/sub implementation instead. We map the event space of both, data plane events (from SA) and control plane events (other controllets), to topics employing a hierarchical topic scheme which allows for fine-grained subscriptions. In the following, we describe the mapping, while illustrating its usage on the example of a SwitchAdapter.

Each controllet defines two sets of *topics*: Set `TO` defines which message types (topics) a controllet is able to process, i.e., which data plane events it wants to receive from the message bus. This set is mapped to corresponding subscriptions for event filtering. Set `FROM` defines the topics published by the controllet, i.e., events disseminated to the bus. Other controllets can subscribe to these advertised topics.

Topic definition is strictly hierarchical. The first hierarchy layer defines the type of declaration (`TO` or `FROM`). The second layer comprises the identity of the controllet. All upper layers contain structure of controllet-type specific content. Attributes are encoded as a bit-sequence, with a specific length associated to each hierarchy layer, at a specific location within the topic-hierarchy. Wildcard matching ("?") is supported.

For the SA, as shown in Listing 1, the semantics are as follows: Listens to Events (TO): The SA will receive any incoming message of these topics and forward it to the switch. Publishes Events (FROM): any OF message the SA receives from the switch is published using a corresponding topic within this set of topics.

Listing 1. Excerpt of the SwitchAdapter topic-hierarchy.

```
TO=0x01                               FROM=0x02
  SWITCH_ADAPTER=0x0000                 SWITCH_ADAPTER=0x0000
    SWITCH_INSTANCE=0x???????             —
      OPENFLOW=0x00                       OPENFLOW=0x00
        FEATURES_REQUEST=0x05               FEATURES_REPLY=0x06
        PACKET_OUT=0x0D                     PACKET_IN=0x0A
        FLOW_MOD=0x0E                         LB_GROUP=0x??  default=0x00
        ROLE_REPLY = 0x19                       IPv4=0x0800
        METER_MOD = 0x1D                          TCP=0x06
                                                  UDP=0x11
                                            PORT_STATUS=0x0C
```

2) Partitioning & Load Balancing: Note that hierarchy layers are not tied to a fixed representation of the underlying event space, e.g., SA topics are not restricted to directly reflect OF-matching fields. Artificial hierarchy layers may be freely introduced between any layers. For instance, to enable load balancing of PACKET_IN messages, the SA artificially discriminates PACKET_INs by introducing an additional 1-Byte topic hierarchy layer (LB_GROUP) and disseminating such events in a round-robin fashion to the set of groups. Controllets participating in load balancing subscribe to a specific LB_GROUP, whereas controllets that want to receive all PACKET_INs apply a wildcard subscription on the LB_GROUP layer. This mechanism enables partitioning along the network topology where, for instance, Topology controllets refine their subscriptions to certain groups.

C. Integration Schemes for LDPEP

One way to implement switch-local control is to identify a set of essential controllets and run them locally on each switch. That way, full modularization is maintained and the controllets' code can be directly reused. While highly scalable, communication over the message bus, e.g., for querying topology data in case of the AFC (see §IV-B1), incurs higher latency compared to, e.g., direct memory access in case of a single-process integration. Although TCP connections over the local loopback interface are highly optimized in recent Linux kernels, micro-benchmarks [13] indicate higher throughput and lower latency when using inter-process communication (IPC).

When focusing on latency, LDPEP should be implemented by a fully integrated, monolithic controllet connected to the message bus in order to leverage the global view and central coordination, as explained for the AFC. Performance potential lays in a tighter coupling to the underlying switch hardware. Ideally, local logic would be pushed down to the data plane hardware using Data Plane Programming, which however focuses on packet processing and thus is not suited to implement arbitrary control logic.

By supporting multiple integration schemes, our architecture offers great flexibility to network operators who have to compromise between performance and implementation efforts, based on the expected load. We have implemented the schemes modularized (*ZSDN-TCP*, *ZSDN-IPC*) and fully integrated (*ZSDN-AFC*) and compare their performance in the following.

VI. Evaluation

In this section, we present the evaluation of our proposed distributed SDN controller architecture, consisting of a raw performance comparison, an analysis of the scalability of our approach, as well as results from the deployment on our white-box networking switch, including the overhead of virtualization.

A. Raw Controller Performance

First, we compare the raw performance of ZSDN with other popular controllers with the following **methodology**.

We use cbench [44] for measuring controller throughput and latency. Cbench emulates switch behavior by sending OF_PACKET_INs (triggers) to the connected controller. To measure throughput, cbench sends triggers as fast as possible and averages over the number of received OF_FLOW_MOD and OF_PACKET_OUT from the controller. To prevent double-counting, we modified the processing of controllers to respond with only one type of message. For sequential throughput, cbench waits for a response to a sent trigger, before sending the subsequent trigger. Hence, we approximate controller processing latency as the inverse of sequential throughput. Cbench and the controllers run on a testbed consisting of 12 nodes (Intel Xeon E3-1245v2 @ 3.4 GHz, 4 physical cores, 16 GB RAM) interconnected through a switched 1 GbE network.

To investigate the impact of controller locality, as illustrated in Figure 6, we differentiate between the switch (cbench) and the controller (traditional SDN controller or ZSDN) running on the same node H_i (*local*; OpenFlow channel: TCP over loopback interface) and running on different nodes H_{i+1} and H_i (*remote*; OF Channel: TCP over switched Ethernet).

Each cbench run is averaging over 60 seconds in parallel on each of the 12 nodes (*local*) and 120 seconds on each of the 6 node pairs (*remote*), totaling in the aggregation of 12 minutes of observation time for each experiment.

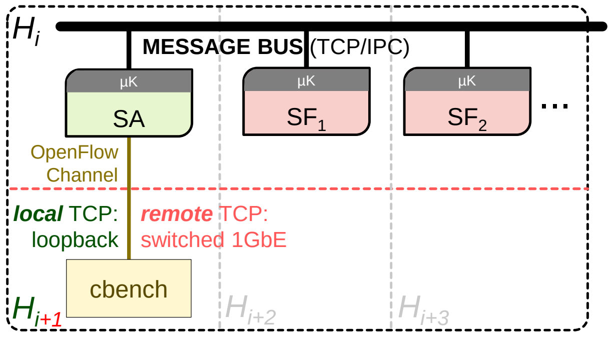


Fig. 6. Evaluation setup of control plane (testbed) for raw controller performance evaluation (§VI-A) and scalability evaluation (§VI-B).

We evaluate the following platforms: (1) ZSDN-TCP/IPC: modular controller framework using single-instance controllets with message bus communication using reliable (guaranteed and in-order message delivery) transport mechanisms TCP and IPC (UNIX domain sockets); (2) ZSDN-AFC: the Autonomous Forwarding LDPEP controllet, as introduced in Section IV-B1, fully integrated (single process, see §V-C); (3) NOX (verity) [45], [46]: an early academic C++ implementation, popular for its performance; (4) ONOS [2]: Java-based, carrier-grade; (5) Floodlight [47]: Java-based, production-grade; (6) Ryu [48]: Python-based, popular for support of recent OF versions.

Figure 7 shows the **results** of the controller comparison, where error bars depict the standard deviation. Regarding *local* **throughput** (Figure 7, top half), NOX performs best with $\approx 369 \pm 2\,\mathrm{msg/ms}$ (messages per millisecond). The LDPEP of ZSDN-AFC results in similar figures with $\approx 260 \pm 1\,\mathrm{msg/ms}$.

The performance penalty of distribution shows to be bearable: distributed ZSDN throughput is about 53% of ZSDN-AFC ($\approx 138 \pm 28\,\mathrm{msg/ms}$), mainly dedicated to message passing. Note that here we ran only one instance of each controllet, thus measuring only the costs of distribution, not its benefits, which we measure in the next section. Interestingly, ZSDN throughput decreases by ⅓ when using IPC instead of TCP. This contradicts expectations risen through the micro-benchmarks, where UNIX domain socket throughput was reported to be about 20% higher than TCP on these nodes. While Floodlight is close to ZSDN-IPC, ONOS performs slightly better. The Python-based controller Ryu is far off with $\approx 0.8\,\mathrm{msg/ms}$. Overall, throughput penalties for a *remote* OF connection are moderate. Interestingly, the remote throughput of ONOS and Floodlight are measured to be higher than their local throughputs, which we could trace down to stem from their common Java-based network framework (Netty). Note that although the control load, i.e., event rate, in practical deployments can be expected to be smaller than in our maximum throughput evaluation, our results provide valuable insights in determining the upper performance bound.

Looking at **latency** (Figure 7, bottom half) however, *remote* latency is increased drastically compared to *local* latency with factors of 2 (ZSDN-TCP) to 6 (ZSDN-AFC). This is a strong argument for local processing, especially for the integrated LDPEP mode. On the other hand, when using modularized controllets, the penalty for running SAs remotely, e.g., for migration or inaccessible control planes (see §IV-D1) is bearable.

B. Scalability of Controllet Distribution

Next, we evaluate the benefits of distribution and replication of ZSDN controllets. As illustrated in Figure 6, we distribute the most compute-intense controllets SwitchAdapter (SA) and SimpleForwarding (SF) to dedicated nodes. For the moment, we use only a single SA instance (replication factor $k{=}1$) placed at H_i (*local*) or H_{i+1} (*remote*). Furthermore, we replicate the SF with a varying replication factor n. Each instance SF_j with $j \in [1, n]$ is placed on a dedicated node (H_{i+1+j}). The SA distributes the total load evenly to these instances (see §V-B2). We additionally vary the number of

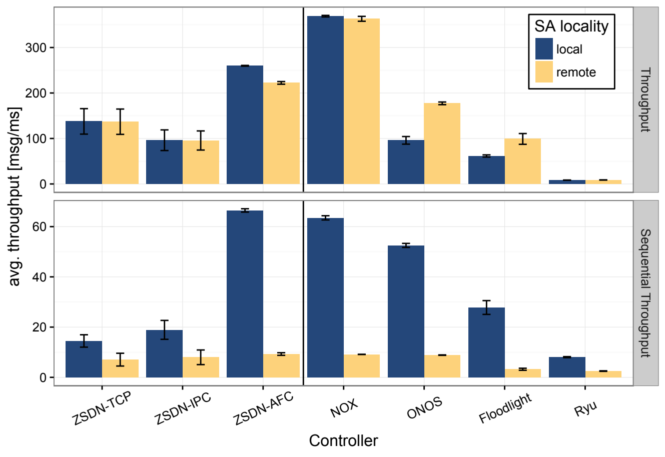


Fig. 7. Raw controller performance: comparison of throughput and sequential throughput (inverse latency) for ZSDN and other popular controllers, drilled down by switch-controller locality (*local*/*remote*).

switches s, cbench emulates. For each switch connection, the SA spawns 4 threads, dedicated to that connection.

The **results** are shown in Figure 8. Even for $n = 1$ (no SF replication), we achieve 15% higher **throughput** just by placing the SF instance on a dedicated node. For $s = 1$ and increasing n, throughput increases, but only sublinearly. In this setting, the SA constitutes a bottleneck. It maxes out 1 thread (per-core performance) and is not able to fire drastically more data plane events which the SF instances could process. If we increase s, the throughput increases almost linearly until the (single) SA instance maxes out (per-CPU performance (all cores)) at $s = 3$. Having $n > 3$ does not further improve performance, such that the overall peak performance is reached with $s = 3, n = 3$ at $\approx 280 \pm 16\,\mathrm{msg/ms}$. Note that maxing out introduces high indeterminism (e.g., apparent throughput drop in graph). To investigate the scaling-up behavior, we repeated the experiment using more powerful nodes (Xeon E5-1650v3 @ 3.5 GHz, 6 physical cores), where throughput peaks at $s{=}5, n{=}5$ with $\approx 670 \pm 20\,\mathrm{msg/ms}$. Note that more compute-intense processing, such as deep packet inspection, would much more benefit from distribution and replication.

For $n = 1$, **latency** increases by about 22% due to the physical separation of SA and SF and thus one additional hop over the control plane network. In contrast to *remote*, *local* latency increases with increasing s, since SA and cbench are running on the same host, thus sharing an increasingly loaded CPU, slowing down cbench's production of emulated data plane events and SA's distribution pace. With increasing n, *local* latency increases as well, due to increased splitting and merging efforts of events to be disseminated to the message bus and reactions received over the bus. *Remote* latency follows the same trend, but only slightly increases with increasing n since the impact on latency of the OpenFlow TCP connection over the physical network instead of the loopback interface is the dominating factor. In more practical scenarios with lower event rates but higher packet sizes, this effect is expected to be of much lesser extent.

When using the full distribution capabilities by replicating both, SF and SA, i.e., increasing n as well as k, and keeping n/k

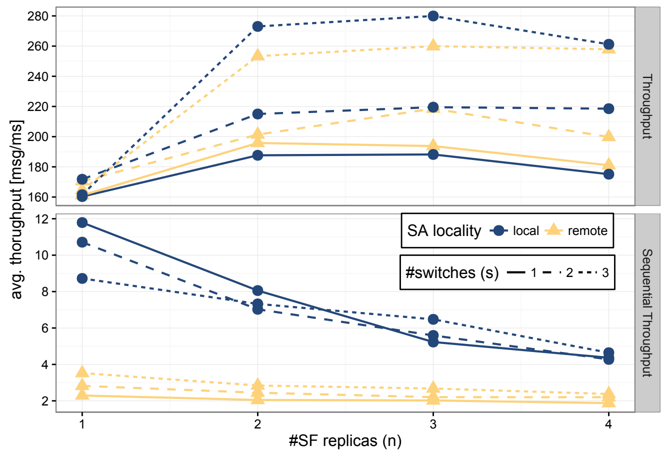


Fig. 8. Scalability evaluation (scale-out): modular ZSDN controller with varying number of SimpleForwarding instances (n) and varying number of connecting switches (s).

balanced, we could verify **linear scalability**. Depending on the efficiency of group communication, which is very efficient in ZMQ due to filtering right at the publisher, network saturation limits scalability. For scenarios with such high event rates however, it is reasonable to employ 10 GbE or higher on the control plane, counteracting network capacity bottlenecks.

C. *Performance on White-box Networking Switch Hardware*

In our last evaluation, we compare the controller performance on real white-box switch hardware, instead of emulating it. We compare ZSDN-IPC, ZSDN-AFC, NOX, and Ryu. The **device under test** is a typical top-of-rack white-box switch Edge-Core AS5712-54X, whose hardware specification is publicly available under the Open Compute Project [38]. Its control plane comprises an x86 Intel Atom CPU with 4 cores at 2.4 GHz, 8 GB RAM, and a 1 GbE NIC. Atop we run the operating systems *OpenNetworkLinux (ONL)* 2.0 with a 3.16.39-LTS kernel and *Pica8 PicOS* 2.8 with a 3.16.7 kernel. On the data plane, it features a Broadcom Trident II ASIC with 48×10 GbE and 6×40 GbE ports.

Our **methodology**, illustrated in Figure 9, is as follows. The switch runs a *network operating system (NOS)* on its control plane. For *remote* performance, the OpenFlow agent running atop of the NOS connects to a remote controller. For *local*, we deploy a controller directly on top the NOS, which the OpenFlow agent connects to. Later, we isolate the local controller using a hypervisor or container. We intentionally provoke that every ingress packet at a data plane port is processed in the control plane. To this end, the controllers run learning switches, but do not install flows. In the data plane, we connect two end-hosts (H_0, H_1) with 10 GbE links to the switch. H_0 is sending packets to H_1 where they are reflected back. Packet identity is ensured through unique sequence numbers attached to the packets (as sole payload). Both, egress (t_{TX}) and ingress (t_{RX}) times are captured using hardware-timestamping. Thus we can measure the RTT at the sender ($T_{RTT} = t_{s,RX} - t_{s,TX}$) and the time spent for reflection at the reflector ($T_{refl} = t_{r,TX} - t_{r,RX}$) with high precision. We approximate the (one-way) switch processing latency as $T_{proc} = {}^1\!/_2 * (T_{RTT} - T_{refl})$, neglecting transmission and propagation delay.

The **throughput results** show significant throttling. This is expected behavior, although the switch silicon is interconnected with the CPU over a PCI-Express bus with plenty of bandwidth. Switch vendors introduce limiting of traffic between the switch silicon (data plane) and the CPU (control plane) in their switch design, to prevent denial of service attacks on the control plane caused by (uncontrolled) data plane traffic. We observe that with ONL, throughput is capped at 1 kpps (1000 packets per second) with a low peak CPU utilization of the ONL's OF-DPA daemon of 50% of one core. This shows that the rate limit is clearly not caused by a CPU bottleneck. For switch-ingress rates ≥ 20 kpps on PicOS, we measure an egress-rate of about 7 kpps, while PicOS's Open vSwitch daemon consumes two cores. Note that due to isolation and resource control, our lightweight virtualization deployment effectively protects control plane operation, rendering additional safety mechanisms like the vendor rate limiting superfluous.

For evaluating **switch control plane processing latency**, we send with a rate of 100 kpps for 50 s. Through reflection, the effective packet rate (ingress rate at the switch) is doubled. We begin with a **comparison of controllers** (Figure 10) running bare-metal (no hypervisor/container) on ONL or remotely (Xeon E5-1650v3).

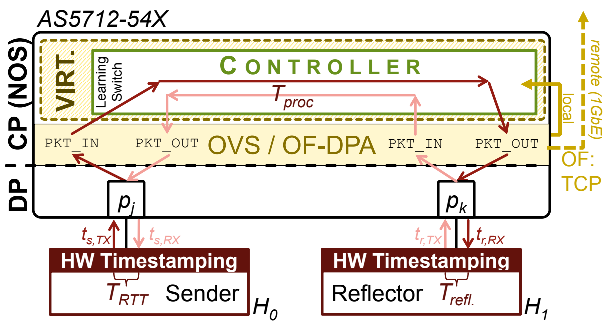


Fig. 9. Setup of measuring control plane processing latency (T_{proc}) on a white-box switch (§VI-C) using hardware-timestamping on end systems.

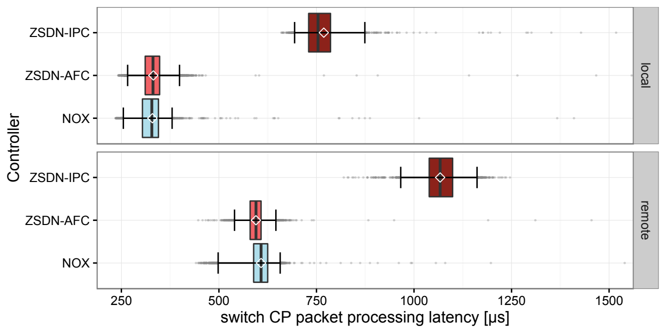


Fig. 10. Evaluation on a white-box networking hardware switch: comparison of switch processing latencies (x-axes: medians (bars), averages (diamonds)) of modular ZSDN-IPC, fully integrated ZSDN-AFC, and NOX, drilled down by switch-controller locality.

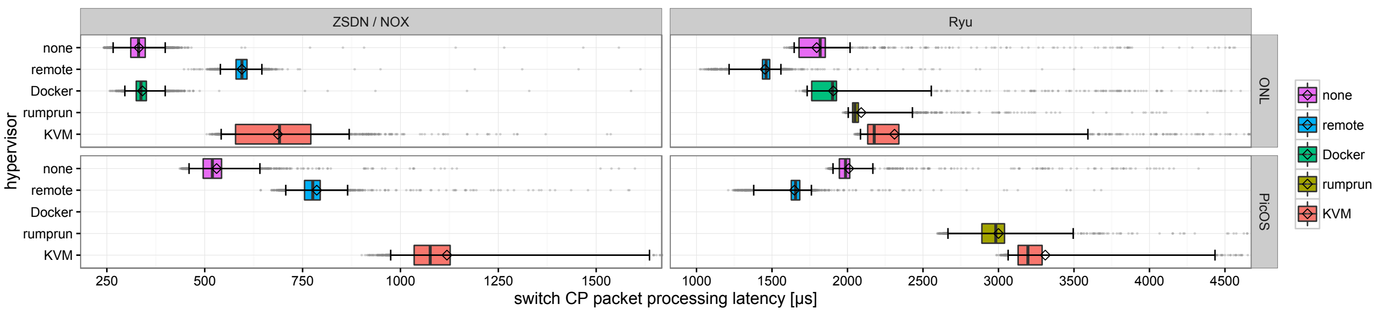


Fig. 11. Processing latencies (x-axes: medians (bars), averages (diamonds)) of controllers (grid horizontal) running locally on network operating systems (grid vertical) with varying isolation mechanisms (y-axis). Bare-metal (*none*) and *remote* execution are given as baselines. Whiskers enclose 95% of measured values.

For switch-*local* controllers, NOX performs best with an average latency of $\approx 330 \pm 75\,\mu s$. ZSDN-AFC is consistently within 3% of NOX. The costs for module decoupling over the message bus of ZSDN-IPC evaluates to $\approx 767 \pm 112\,\mu s$, a factor of 2.3, clearly showing the superiority of LDPEP with the full integration scheme of ZSDN-AFC which still profits fully from centralized view and coordination. *Remote* execution increases latency by a factor of ≈ 1.8, for ZSDN-AFC and NOX and a factor of ≈ 1.4 for ZSDN-IPC. Note that our scenario of a one-hop switched 1 GbE control network is almost ideal, providing a lower bound for switch control plane processing latency. For larger distances or WAN scenarios, remote control latencies are expected to be orders of magnitudes higher. Ryu (Figure 11) is expectedly performing worse than its C++ counterparts with $\approx 1795 \pm 225\,\mu s$ but surprisingly 20% faster remote execution.

We evaluate **virtualization overhead** of *Docker* (ONL only) as well as rump kernels (*rumprun*) and full VMs (*KVM*), both running on QEMU with KVM-acceleration enabled by the Atom's VT-x support. The baseline is bare-metal execution (*none*). Since NOX and ZSDN are relying on Digital Shared Objects (DSO), we were not able to port them to rumprun. The results are illustrated in Figure 11. **Docker** has the lowest overhead. With full isolation of all but the network namespace, Docker imposes almost no overhead for NOX/ZSDN. Latency and its deviation are within $1\,\mu s$ to bare-metal execution.

Next, we measure the combined overhead of the hypervisor and the guest OS of virtualization variants, employing KVM's pseudo-paravirtualized network driver *virtio*. **Full virtualization** (*KVM*) adds large overhead. On average, $410\,\mu s$ (factor 1.5) incur for ONL and $820\,\mu s$ (factor 1.8) for PicOS, both slower than remote execution. Standard deviations are larger by factors 2 and 2.2, respectively. Ryu as a **Unikernel** (*rumprun*) is showing much better results. Compared to a full VM, latencies and deviations are greatly reduced by $220\,\mu s$ and $285\,\mu s$ for ONL, and $310\,\mu s$ and $190\,\mu s$ for PicOS. This is the result of the minimal guest OS and hence reduced OS overhead. Compared to bare-metal/Docker, factors of 1.2 and 1.5 for latency on ONL and PicOS are promising.

Lastly, we evaluate the difference between the **NOSes**. For all measurements, compared to ONL, PicOS adds quite consistent latency of $200\,\mu s$ on average for bare-metal and remote, $420\,\mu s$ for NOX/ZSDN, and as high as $950\,\mu s$ for KVM and rumprun. Especially for higher packet rates, we have observed instability of QEMU on PicOS.

We can conclude that containers provide isolation as needed at minimal cost. We could verify and quantify the benefit of reduced latency to be almost halved with containerized local control logic, despite isolation.

VII. Related Work

Many early approaches, including Onix [49], propose to externalize state storage, which incurs additional latency for lookups. In Onix, switches and controller instances are tightly coupled. While Onix limits the shared view onto network state information, HyperFlow [50], as our approach, holistically propagates all kinds of data plane events. HyperFlow also facilitates pub/sub to propagate events, event classification is however limited to three topics, whereas our approach leverages content-based filtering (mapped to a topic hierarchy in our preliminary implementation) to allow for fine-grained subscriptions. Furthermore, HyperFlow exclusively relies on passive synchronization of the locally cached network wide view, while our approach offers maximum flexibility allowing both, local caches for fast access as well as access to highly consistent centralized storage.

DevoFlow [5] is the first SDN approach to allow for local decision making on the switch, however mandating changes of the switching ASIC. Kandoo [6] proposes a two-layered controller hierarchy with a root controller maintaining network-wide state, and local controllers possibly running directly on switching hardware, only handling local events where no global knowledge is required. While this scheme allows for offloading of simple local logic, local controllers do not hold any state data, neither do they interact with each other at all. Our approach is not limited to such a strict hierarchical scheme and does not rely on a root controller instance, thus offering superior flexibility.

While these approaches exhibit a static switch-controller assignment, ElastiCon [7] allows for a dynamic switch to controller instance mapping. By periodic monitoring of controller load, the number of instances and the mapping is adapted for effective load balancing. Since switches are still tightly coupled to an instance, the authors introduce a switch migration protocol. A similar problem is addressed in [10], [11]. The decoupling of switch and controller offered by our approach eliminates the need for complex and costly migration mechanisms.

More recent approaches improve on failure tolerance in control distribution. Beehive [51] models control applications as centralized asynchronous message handlers featuring and thus focusing on application partitioning, exclusive handling of messages among a set of controllers, as well as consistent replication of control state information. Logical message propagation is dictated by map-functions that determine to which set of applications a specific message is to be sent to. Message passing is not addressed in detail. Furthermore, each Beehive controller instance contains all application logic in contrast to our highly modular approach. Another work, Ravana [17], focuses on controller fault-tolerance. Ravana subsumes event dissemination from switches, their processing by a controller, and the resulting execution of controller commands at the switches in a transaction and guarantees that control messages are processed transactionally with exactly-once semantics. Message propagation and actual distribution schemes are not addressed.

Fibbing [52] exerts centralized control over routers that implement a legacy, non-SDN, control plane running fully decentralized routing algorithms, such as OSPF and IS-IS. The forwarding behavior of routers, i.e., their forwarding information base, is manipulated as to achieve desired network behavior by faking input messages to the distributed routing algorithms. Although being congruent in the notion of centralized control, unlike in our approach, Fibbing's control is solely indirect and thus inherently limited.

VIII. Conclusion

In this article, we presented a novel architecture for a highly flexible distributed SDN controller based on a message bus for communication and a micro-kernel design. Network control logic is split into control modules, called controllets, which can be freely distributed. Controllets communicate through the message bus and are decoupled from switches and other controllets using the publish/subscribe paradigm. The micro-kernel design only requires controllets to implement a small set of functions to connect to the message bus and participate in publish/subscribe communication. Consequently, controllets are extremely lightweight and can also be executed directly on white-box switches to enable fully distributed network control even without external SDN controller—a new level of flexibility in control plane distribution that so far is not possible with standard SDN controllers. Our evaluations showed the practicality of our architecture for both, full distribution as well as the integration of controllets for fast local processing of data plane events, while still benefiting from global view and centralized coordination. Through employing lightweight virtualization techniques, we cope with crucial challenges of practical deployment to ensure a safe operation of the control plane and thus continuous network operation.

Possible future work includes evaluating specific patterns of data plane events and control plane events to complex network events as well as an event-driven execution of system management workflows in a holistic system management plane embracing the network control plane.

References

[1] Open Networking Foundation, "OpenFlow Switch Specification," https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.3.4.pdf.

[2] The Linux Foundation, "ONOS - A new carrier-grade SDN network operating system designed for high availability, performance, scale-out," http://onosproject.org/.

[3] OpenDaylight Foundation, "OpenDaylight: Open Source SDN Platform," https://www.opendaylight.org/.

[4] OSGi Alliance, "Open Services Gateway initiative," https://www.osgi.org/.

[5] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling Flow Management for High-performance Networks," in *Proceedings of the ACM SIGCOMM 2011 Conference*, ser. SIGCOMM '11. New York, NY, USA: ACM, 2011, pp. 254–265.

[6] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12. New York, NY, USA: ACM, 2012, pp. 19–24.

[7] A. A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, "ElastiCon: An Elastic Distributed Sdn Controller," in *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '14. New York, NY, USA: ACM, 2014, pp. 17–28.

[8] R. Bifulco, J. Boite, M. Bouet, and F. Schneider, "Improving SDN with InSPired Switches," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '16. New York, NY, USA: ACM, 2016, pp. 11:1–11:12.

[9] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 44–51, Apr. 2014.

[10] A. Krishnamurthy, S. P. Chandrabose, and A. Gember-Jacobson, "Pratyaastha: An Efficient Elastic Distributed SDN Control Plane," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14. New York, NY, USA: ACM, 2014.

[11] A. Basta, A. Blenk, H. B. Hassine, and W. Kellerer, "Towards a dynamic SDN virtualization layer: Control path migration protocol," in *2015 11th International Conference on Network and Service Management (CNSM)*, Nov. 2015, pp. 354–359.

[12] D. Chappell, *Enterprise service bus*. " O'Reilly Media, Inc.", 2004.

[13] T. Kohler, F. Dürr, and K. Rothermel, "ZeroSDN: A Highly Flexible and Modular Architecture for Full-range Network Control Distribution," in *2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE Press, May 2017, pp. 25–37.

[14] T. Kohler, F. Dürr, C. Bäumlisberger, and K. Rothermel, "InFEP - Lightweight Virtualization of Distributed Control on White-box Networking Hardware," in *2017 13th International Conference on Network and Service Management (CNSM)*, Nov 2017, pp. 1–6.

[15] ZSDN, "Github Repository: Zero Software Defined Networking; Distributed Software Defined Networking (SDN) Controller," https://github.com/zeroSDN.

[16] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, Jun. 2003.

[17] N. Katta, H. Zhang, M. Freedman, and J. Rexford, "Ravana: Controller Fault-tolerance in Software-defined Networking," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15. New York, NY, USA: ACM, 2015.

[18] S. Bhowmik, M. A. Tariq, B. Koldehofe, F. Dürr, T. Kohler, and K. Rothermel, "High Performance Publish/Subscribe Middleware in Software-Defined Networks," *IEEE/ACM Transactions on Networking*, vol. PP, no. 99, pp. 1–16, 2017.

[19] nanomsg Documentation. (2017, Jan.) Differences between nanomsg and ZeroMQ. http://nanomsg.org/documentation-zeromq.html. [Online]. Available: http://nanomsg.org/documentation-zeromq.html

[20] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat, "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 183–197.

[21] 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, ser. HotSDN '12. New York, NY, USA: ACM, 2012, pp. 1–6.

[22] 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*, ser. HotSDN '13. New York, NY, USA: ACM, 2013, pp. 121–126.

[23] C. Cascone, L. Pollini, D. Sanvito, and A. Capone, "Traffic Management Applications for Stateful SDN Data Plane," in *2015 Fourth European Workshop on Software Defined Networks*, Sep. 2015, pp. 85–90.

[24] A. S. Muqaddas, P. Giaccone, A. Bianco, and G. Maier, "Inter-controller traffic to support consistency in onos clusters," *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 1018–1031, Dec 2017.

[25] VMWare Inc., "Open vSwitch – An Open Virtual Switch (Nicira Extensions)," https://git.io/vgTKL.

[26] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The Design and Implementation of Open vSwitch," 2015, pp. 117–130.

[27] J. Yang, Z. Zhou, T. Benson, X. Yang, X. Wu, and C. Hu, "Focus: Function offloading from a controller to utilize switch power," 2016.

[28] N. L. M. v. Adrichem, B. J. v. Asten, and F. A. Kuipers, "Fast Recovery in Software-Defined Networks," in *2014 Third European Workshop on Software Defined Networks*, Sep. 2014, pp. 61–66.

[29] D. Thaler and C. Hopps, " Multipath Issues in Unicast and Multicast Next-Hop Selection," Internet Requests for Comments, RFC Editor, RFC 2991, November 2000. [Online]. Available: https://tools.ietf.org/html/rfc2991

[30] Broadcom, "BroadView: Analytics-Driven Dynamic Path Optimization," https://www.broadcom.com/collateral/tb/BroadView-TB301-RDS.pdf.

[31] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid, "A Distributed and Robust SDN Control Plane for Transactional Network Updates," in *Proceedings of INFOCOM'15)*, Apr. 2015.

[32] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-independent Packet Processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[33] T. Kohler, R. Mayer, F. Dürr, M. Maaß, S. Bhowmik, and K. Rothermel, "P4cep: Towards in-network complex event processing," in *Proceedings of the ACM SIGCOMM 2018 Morning Workshop on In-Network Computing*, ser. NetCompute '18. New York, NY, USA: ACM, 2018, pp. 33–38. [Online]. Available: http://doi.acm.org/10.1145/3229591.3229593

[34] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-Network Computation is a Dumb Idea Whose Time Has Come," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XVI. New York, NY, USA: ACM, 2017, pp. 150–156.

[35] A. Bremler-Barr, Y. Harchol, and D. Hay, "OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 511–524. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934875

[36] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "Netbricks: Taking the v out of NFV," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, 2016, pp. 203–216. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda

[37] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: A Framework for NFV Applications," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: ACM, 2015, pp. 121–136.

[38] Open Compute Project. Networking Specs And Designs. https://www.opencompute.org/wiki/Networking/SpecsAndDesigns.

[39] Markets and Markets. https://www.marketsandmarkets.com/Market-Reports/white-box-server-market-219773580.html.

[40] A. Kantee and J. Cormack, "Rump kernels: No os? no problem!" in *;Login: USENIX Magazine, October 2014, Vol. 39, No. 5*. USENIX.

[41] iMatix Corporation, "ZeroMQ: Distributed Messaging," http://zeromq.org/.

[42] iMatix Corporation. CurveZMQ - Security for ZeroMQ. http://curvezmq.org/page:read-the-docs.

[43] Google Inc., "Protocol Buffers," https://developers.google.com/protocol-buffers/.

[44] R. Sherwood and K.-K. Yap, "Cbench: an Open-Flow Controller Benchmarker," http://www.openflow.org/wk/index.php/Oflops.

[45] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an Operating System for Networks," *SIGCOMM CCR*, vol. 38, no. 3, pp. 105–110, Jul. 2008.

[46] The NOX Controller, "Github Repository: NOX Network Control Platform," https://github.com/noxrepo/nox.

[47] Big Switch Networks, "Floodlight: An Open SDN Controller," http://www.projectfloodlight.org/floodlight/.

[48] Ryu SDN Framework Community, "Ryu: Component-based Software Defined Networking Framework," https://osrg.github.io/ryu/.

[49] 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 *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI '10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6.

[50] A. Tootoonchian and Y. Ganjali, "Hyperflow: a distributed control plane for openflow," in *Proceedings of the 2010 internet network management conference on Research on enterprise networking*. USENIX Association, 2010, pp. 3–3.

[51] S. H. Yeganeh and Y. Ganjali, "Beehive: Simple distributed programming in software-defined networks," in *Proceedings of the Second ACM Symposium on SDN Research (SOSR '16)*, Santa Clara, CA, Mar. 2016.

[52] S. Vissicchio, O. Tilmans, L. Vanbever, and J. Rexford, "Central Control Over Distributed Routing," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 43–56.

Thomas Kohler received the M.Sc. degree in Computer Science from Augsburg University, Germany, in 2013. He is currently pursuing the Ph.D. degree at the Distributed Systems research group, University of Stuttgart, Germany. His research interests include consistency and determinism in Software-defined Networking as well as White-box and programmable networking hardware. In particular, his research focusses on update consistency, local switch logic, control plane distribution, and data plane programming.

Frank Dürr is a senior researcher and lecturer at the Distributed Systems Department of the Institute of Parallel and Distributed Systems (IPVS) at University of Stuttgart, Germany. He received both his doctoral degree and diploma in computer science from University of Stuttgart. Frank Dürr is currently leading the mobile computing and the software-defined networking (SDN) & time-sensitive networking (TSN) groups of the Distributed Systems Department. He has given tutorials on SDN at several national and international conferences, and as a lecturer he has been giving lectures and practical courses on networked systems and SDN. Besides SDN and TSN, Frank Dürr's research interests include mobile and pervasive computing, location privacy, and cloud computing aspects overlapping with these topics like mobile cloud and edge computing, or datacenter networks.

Kurt Rothermel received his doctoral degree in Computer Science from University of Stuttgart in 1985. From 1986 to 1987 he was Post-Doctoral Fellow at IBM Almaden Research Center in San José, U.S.A., and then joined IBM's European Networking Center in Heidelberg. Since 1990 he is a Professor for Computer Science at the University of Stuttgart. From 2003 to 2011 he was head of the Collaborative Research Center Nexus (SFB 627), conducting research in the area of mobile context-aware systems. He is a Director of the Institute of Parallel and Distributed Systems. His current research interests are in the field of distributed systems, computer networks, and mobile systems.