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

Thomas Kohler Frank Dürr Kurt Rothermel
{firstname}.{lastname}@ipvs.uni-stuttgart.de
University of Stuttgart
Institute of Parallel and Distributed Systems (IPVS)
Universitätsstraße 38
70569 Stuttgart

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 in this paper, providing maximum flexibility with respect to distribution.

Our architecture splits control logic into light-weight 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. Light-weight controllets also allow for pushing control logic onto switches and enable local processing of data plane events to minimize latency and communication overhead. Controllets are interconnected through a message bus supporting the publish/subscribe communication paradigm with specific extensions for content-based OpenFlow message filtering. Publish/subscribe allows for complete decoupling of controllets to further facilitate control plane distribution. We evaluate both, the scalability and performance properties of our architecture, including its deployment on a white-box networking hardware switch.

CCS Concepts

• **Networks** → **Network architectures;** *Programmable networks; Bridges and switches; Network manageability;* • **Software and its engineering** → **Publish-subscribe / event-based architectures;**

Keywords

Software-defined Networking; OpenFlow; Control Plane Distribution; Publish/Subscribe; White-box Networking

Published in ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS'17, May 18-19, 2017, Beijing, China

1. INTRODUCTION

Software-defined Networking (SDN) is based on the paradigm of *logically centralized control* of network elements. Logical centralization is nothing more than 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.

1.1 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 [28], 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* (Fig. 1(a)), where control modules implementing certain control functions can be dynamically (un-)loaded into the controller process at runtime. One popular example showing that this design is still frequently used in practice is the OpenDaylight [29] controller, relying on OSGi [30]. However, this modular controller architecture remains to be monolithic since it still relies 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* (Fig. 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. Note, that this partitioning over control functions, as depicted, mandates multiple concurrent controller connections. This feature was originally unsupported by OpenFlow and has been added in version 1.2. However, vendor-specific implementation in hardware switches might impose restrictions, e.g., on the number of concurrent controller connections [18], limiting practicability. To overcome compatibility issues, an $n{:}m$ switch-controller-mapping can be implemented by a simple wrapper[1] that runs locally on each switch and proxies OpenFlow messages from the switch to control modules, and vice-versa. Secondly, control can be partitioned over the network topology, i.e., the scope of individual control modules can be restricted to a subset of switches. This possibly requires further concepts to coordinate instances with different scopes, e.g., through a controller hierarchy, on the other hand facilitates scalability with the network size.

1.2 Motivation: A Full-range Distribution Architecture for SDN Controllers

Observing that distributed SDN controllers already exist today, can we conclude that SDN controller 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. Thus, with current SDN, switches talk to the external monolithic network controller or distributed external control modules, but not to other switches. There is no direct communication between network elements. This reflects the clean-slate paradigm shift from distributed network control to logically centralized control, where switches are just fast forwarders and all "intelligence" is outsourced to external machines. Obviously, this outsourcing comes at a cost like increased round-trip times (slower reaction), increased control network load, 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: we must enable the possibility to bring control back onto the switch. Although execution of control logic on the switch hardware on the one hand has been conceptually proposed in literature [12, 17, 13], 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 [4] or state machine logic [3]. To fully exploit locality of the switches, we argue to include the switch hardware 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 latency*, while leveraging the logically centralized paradigm of SDN through access to global knowledge, i.e., access to neighbor switches as well as to the entire controller hierarchy, in order to *improve decision quality*. Since requirements, such as timeliness, optimality, and consistency, may tremendously differ between network functions, a network control architecture should provide the flexibility for balancing these trade-offs for each individual network function to account for their heterogeneity. For instance, for forwarding decisions at a switch, which are inherently distributed, typically full global knowledge is not required but the focus lays on timeliness in order to reduce network 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.

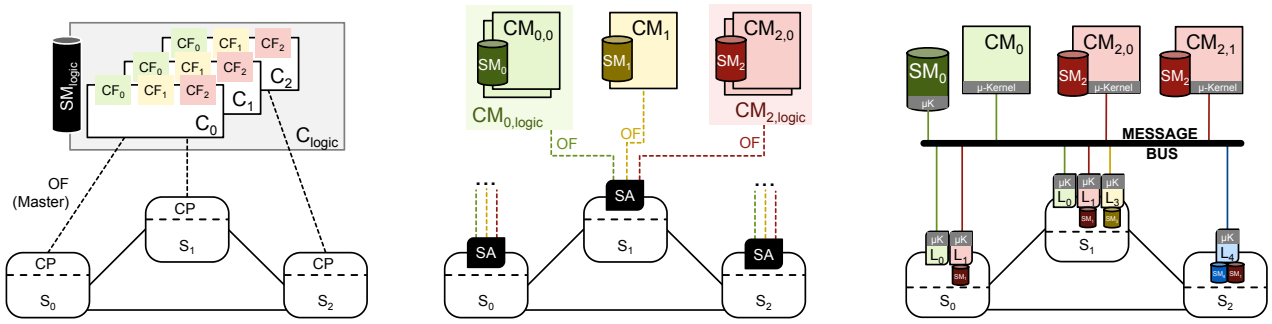
Secondly, with the current concept we observe that controllers tend to be quite heavy-weight (which might also be a practical reason why control is removed from switches). For instance, in order to just receive packet-in events, the prominent OpenDaylight controller requires a full-fledged OSGi environment with a total code size of $\approx 280\,\text{MByte}$. More light-weight controllers 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, basically 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 light-weight *micro-kernel* approach for SDN controllers instead of a heavy-weight 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 [23, 2]. 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 [11]. 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.

The **main contribution** of this paper 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 full spectrum from logically centralized to fully distributed control; (2) micro-kernel controller architecture for distributed light-weight 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. An implementation of the proposed concepts is publicly available on GitHub [47].

The rest of the paper is structured as follows. In §2, we describe the architecture of our distributed SDN controller together with an overview of the basic concepts. We proceed with describing the message bus concept in more detail

[1] denoted as SwitchAdapter (SA) in our implementation



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

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

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

Figure 1. Evolution of distribution in SDN controller architectures. Rightmost: Our envisioned fully distributed architecture.

in §3. In §4, we discuss how this concept enables highest flexibility in terms of control plane distribution, before we present detailed applications for executing local logic based on global knowledge, which we denote as local data plane event processing. In §5 we describe the most important aspects of our implementation of the concepts, followed by an evaluation of performance and scalability of our distributed control architecture as well as results from the deployment of local logic on white-box networking switch hardware in §6. Based on the presented concepts for a distributed network controller, we outline a roadmap towards a highly scalable holistic system control plane in §7, before we discuss related work in §8, and conclude the paper in §9.

2. ARCHITECTURE

We start by introducing the basic architecture of our distributed SDN controller (cf. Fig. 1(c)) including an overview of the basic functions and concepts.

Our approach is based on what we call a *micro-kernel architecture* for SDN controllers. We split network control logic into light-weight control modules, whose instances we call *controllets* (CM_i). In contrast to a monolithic controller, controllets do not require a heavy-weight execution environment. Instead we execute each controllet in a separate process, possibly being also physically distributed to separate hardware, and enable each controllet to communicate with other controllets or switches through messages. 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 light-weight nature of controllets also enables us to execute controllets directly on switches (S_i), typically featuring limited computing resources, to push-down control logic onto switches. This decreases communication latency and overhead. We denote controllets running locally on switches as L_i.

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. Controllets can run on any hardware entity connected to the message bus, e.g., switches, or server hardware. Each controllet and switch can send messages through the bus to other controllets or switches either using the request/response or the publish/subscribe (pub/sub) paradigm. Decoupling controllets and switches allows for flexible distribution including migration of controllets, and dynamic spawning or exchanging of controllets at runtime.

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.

3. THE SDN MESSAGE BUS: DECOUPLING CONTROLLETS THROUGH EVENTS

Our architecture utilizes *event-based communication* to decouple the producers of events from the consumers of events in time (asynchronous communication) and space (distribution of logic between nodes including switches and hosts). 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, such as physical network topology knowledge and end-host protocol state, e.g., TCP-sessions, can be inferred. 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, or is being forwarded to the control plane, e.g., when no matching forwarding rule exists. In this case, the switch ASIC passes the associated packet to the OpenFlow agent (running on the switch's CPU), which itself encapsulates the packet into an OpenFlow `PACKET_OUT` message. When not consumed locally (c.f. §4.2), 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 [14]. Therefore, event conditions include matches on header field tuples or any other meta-data. This paradigm can also emulate standard client/server communication, multicast, or topics [14], using filters on receivers, groups, topics, etc.

However, we do not restrict ourselves to *basic data plane events*, but also consider *complex data plane events* involving, for instance, several 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 don't further elaborate on complex data plane events in this paper.

Another type of events used for inter-controllet communication is the control plane event (CPE), which encapsulates 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.

Recent SDN research has shown that consistency in an inherently distributed system of switches and controllers might require certain semantics on the delivery of messages ("exactly once" delivery) [21]. The message bus paradigm is well-suited to transparently implement this strong semantic together with more light-weight semantics like "at most once" delivery for less critical tasks. The necessary code for the publishers and subscribers is part of the micro-kernel included by every controllet.

Since the message bus depicts a crucial system component, we would like to briefly discuss its implications regarding scalability and reliability. Modern brokerless message bus software implementations use efficient transport mechanisms for event dissemination, like multicast or unicast with publisher-side subscription based filtering, targeting scalability to hundred thousands of subscriptions [24], which suffices to accommodate typical data center networks [26, 37]. Furthermore, 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. We will address this concept in future work. Failure of the message bus translates to a broken control channel which is equally 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.

Events are heavily used by distributed control, which uses the message bus for both, control coordination and state data synchronization, as described next.

4. HIGHLY FLEXIBLE CONTROL PLANE DISTRIBUTION

Common SDN architectures have been reducing switches to "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 control from the switch increases latency due to incurring switch-controller round-trip times and network overhead due to remote communication. In this section, we will show how light-weight controllets can bring back control onto the switch while benefiting from the logically centralized paradigm of SDN, while also addressing drawbacks of control decentralization.

4.1 Augmented Fully Distributed Control

SDN has abandoned fully decentralized network control based on a distributed control plane implemented solely by switches in favor for logically centralized control. We do not want to strictly argue for or against logically centralized control or fully distributed control. However, we observe that the strict notion of separating data plane elements and the logically centralized control plane imposed by OpenFlow 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 [21], vigorous efforts have to be undertaken to provide the same fault-tolerance with a logically centralized SDN network. Therefore, we stress the fact that maintaining a global view and applying logically centralized control algorithms comes at a cost, and the advantages of logically centralized control should be traded-off well against its disadvantages. Consequently, we argue that truly flexible network control also includes the option for full distribution of network control, to let the network operator decide what paradigm fits his needs best.

Moreover, recent developments in networking hardware make it feasible to push control logic onto switches, due to a) increased computing performance and b) programmability through open access to the switch's control plane[2]. *White-box networking switches* feature open, Linux-based network operating systems (NOS) as the control plane, running on increasingly powerful CPUs (c.f. §6.3).

Therefore, and in-line with recent research [35, 3, 4, 10], our architecture supports pushing light-weight 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 using events—thus, we can implement distributed network control alongside logically centralized network control, or implement anything in-between. This scheme allows for the best of both worlds—fully decentralized processing, yet being centrally coordinated, and logical centralization, which allows for gauging trade-offs like timeliness w.r.t. event processing or convergence time for classic decentralized control algorithms against distribution and synchronization overheads. Furthermore, through incorporation of (more) global knowledge available through logical centralization, we can additionally trade-off against solution optimality of control decisions. As we show in the following use cases, solely 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 upfront 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 or aggregate 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.

[2]We also observe a trend for increasing access to data plane programmability. For instance, for popular switching ASICs, Broadcom offers the *OpenFlow - Data Plane Abstraction*[8] implementing an adaptation layer between OpenFlow and the Broadcom ASIC SDK, while being generalized to provide a vendor-independent access to switching ASICs in a uniform manner by the *Switch Abstraction Interface* [33].

4.2 Local Data Plane Event Processing

We argue to place 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). Exploiting locality allows for reacting most timely on data plane events, thus not only decreasing latency, but also being able to decide on most recent state data. Furthermore, opposed to a non-local controllet, the total control load is inherently balanced to local control modules relieving the message bus.

In LDPEP, 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 present essential use cases enabled by LDPEP.

4.2.1 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* [42] in the prominent SDN software switch implementation Open vSwitch (OVS) [32].

In the following we will present the concept of *Autonomous Forwarding*, which is illustrated in Figure 2, running on a typical switch hardware platform. Following standard OpenFlow behavior, packets (❶ from $Host_{src}$ destined to $Host_{dst}$) without matching forwarding rules ❷ are escalated 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 to be the end host MAC to switchport 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. Also, forwarding decisions might violate global network policies, such as firewall rules, ACLs, or tenant isolation.

To implement centrally coordinated control, preventing policy conflicts, and leverage global network view, the Autonomous Forwarding controllet (AFC) thus 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 switchports 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.

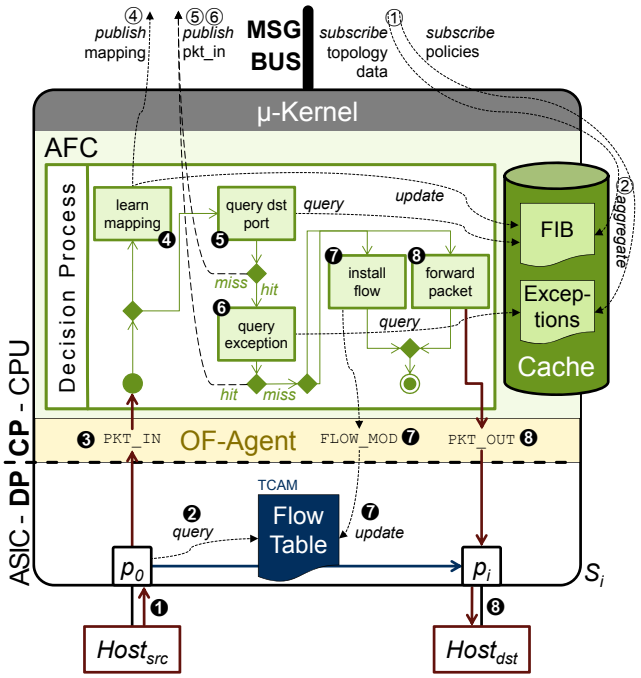


Figure 2. Schematic overview of the Autonomous Forwarding controllet and its processing of a local data plane event (forwarding of $Host_{src} \rightarrow Host_{dst}$) on a typical switch hardware platform.

In the forwarding decision process, the MAC-switchport 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 is not solely needed by other AFCs, but is highly relevant for many other controllets, it is thus published to the message bus ④. Then, the cached topology data, i.e. the FIB, is queried for the switchport associated with $Host_{dst}$ ❺. In case the data is not present locally, the `PACKET_IN` 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 check 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.

While maintaining an exception list locally is mandatory, 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 could just store the primary source for that data—

the peer (controllet) at which the data is local. Thus, in case such data is needed, the respective peer could be queried directly rather than publishing an uninformed query to the whole message bus.

4.2.2 ARP Handling

ARP is another essential networking mechanism, which has already been investigated in the context of local control and controller-function offloading [4, 43]. 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, rather than a proactive scheme where forwarding rules keep the (known) non-control flows in the data 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 aforementioned literature.

4.2.3 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 a first example, **local fast failover**, a link failure—just 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[3] to belong to a redundant path to S_2 (which is typical for, e.g., data center network topologies). 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 finally 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-peer communication, and adapt it in case of discovered sub-optimality. A related approach [1], 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, resteering 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) [38]. Recent switch instrumentation technologies, like Broadcom's BroadView [7], even enable fine-grained access to hardware switchport queue statistics, which allows for much 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 modern data center network topologies.

[3]Switch to switch links can be discovered by employing active probing using LLDP [19], as described in §5.

4.2.4 Control Plane Feedback Mechanism

While not strictly processing data plane events, there exists another appealing use case, enabled by local logic: rules in a switch's flow table constitutes state data as well, however control plane state rather than data plane state. Changes to these rules therefore depict a type of control plane event. Since local controllets are the only entities that can *directly* access the switch's flow table entries, any applied change to a flow table can be propagated to interested controllets. This implements a feedback mechanism that allows an issuing controllet to verify whether its flow change has been successfully applied—a precursor for a transactional interface [9]. 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.

4.2.5 Migration and Closed Switch Hardware

If a switch allows for local logic execution, we can scale the scope of data it caches according to its available resources, as described earlier. If a switch is not powerful enough to execute a controllet or the switch's control plane is inaccessible, we provide a fallback mechanism which still 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 a direct OpenFlow connection and acts as a gateway to and representation of the switch in the message bus. Note, that logically, an external SwitchAdapter is still capable of executing local logic, yet additional network latency is incurred. In our evaluations (c.f. §6), we determine the penalty of externalizing the SwitchAdapter (c.f. §5.2).

5. IMPLEMENTATION

We have implemented a prototype of our distributed SDN controller architecture, consisting of a modular execution framework (*ZMF* [46]) running a distributed SDN controller application (*ZSDN* [47]) with essential controllets atop. ZMF and most modules are written in C++ , however, we provide a Java-based module framework (*JMF* [45]), as well as exemplary modules in Java. Both feature build support for x86 and ARM architectures. In this section we will present the most important aspects of our implementation.

5.1 The Zero Module Framework

Module runtime environments are completely decoupled and independent of each other (not considering business logic dependencies). They run in own processes, possibly on separate hardware. The ZMF implements automatic module discovery along with logical dependency and life-cycle management (`PeerDiscoveryService`), enabling peer dynamics. Modules (controllets) can join or leave the framework at runtime, causing dependent modules to stop/resume operation. Module lifecycle state data is propagated to other modules

using UDP multicast. Thus, each module knows all other modules and their state.

For the message bus we employ a production-grade low-latency communication middleware *ZeroMQ* (ZMQ) [20]. ZMQ bindings are available for all major programming languages. Later, we will show the mapping of both data plane events and control plane events to pub/sub topics. We use TCP and IPC as ZMQ's underlying transport mechanisms. Access to the message bus is provided to ZMF modules through the `MessagingService`. Our micro-kernel implementation comprises both, the `PeerDiscoveryService` and `MessagingService`.

5.2 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* [15] definitions, providing language-independent module communication.

Figure 3 shows essential controllets and their logical interdependencies. The SwitchAdapter (SA) wraps an OF-enabled switch in an instance which is running locally on the switch, integrating it to the framework. An OF switch connects to an SA instance as it would normally connect to a controller. From the perspective of the switch, its SA is its controller. From the perspective of any other controllet, an SA instance represents an OF switch.

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 [19]) 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 hosts, 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. For instance, Forwarding and SimpleForwarding controllets subscribe to `PACKET_IN` data plane events and process them by installing forwarding rules or forwarding the underlying packet to its destination switchport via a `PACKET_OUT`.

5.2.1 Event Space – Topics Mapping

Due to the lack of usable 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*:

1) 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. Note, that technically, ZMQ maps pub/sub to unicast TCP, where filtering is applied on the publisher side, such that unnecessary control traffic is avoided.

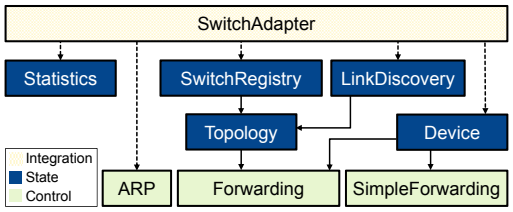


Figure 3. Dependency graph for essential controllets.

2) 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, such that, for instance, controllets can have flows installed by firing a `FLOW_MOD` event. 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
  SWITCH_ADAPTER=0x0000
    SWITCH_INSTANCE=0x????????????????
      OPENFLOW=0x00
        FEATURES_REQUEST=0x05
        PACKET_OUT=0x0D
        FLOW_MOD=0x0E
FROM=0x02
  SWITCH_ADAPTER=0x0000
    OPENFLOW=0x00
      FEATURES_REPLY=0x06
      PACKET_IN=0x0A
        LB_GROUP=0x??  default=0x00
          IPv4=0x0800
            TCP=0x06
            UDP=0x11
          ARP=0x0806
      PORT_STATUS=0x0C
      FLOW_MOD=0x0E
```

5.2.2 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_IN`s 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_IN`s apply a wildcard subscription on the `LB_GROUP` layer. This mechanism can also be

used for partitioning along the network topology where, for instance, Topology controllets refine their subscriptions to certain groups. Technically, we declare topic definitions in a structured format in dedicated files for language-agnostic access and ease of use.

5.3 Integration schemes for LDPEP

For implementing LDPEP, different integration schemes exist:

To maintain the full modularization and to fully reuse controllets code, a set of controllets implementing LDPEP can be identified, like in the distributed case, but being run locally on the switch, instead of being physically distributed. While highly scalable, communication over the message bus, e.g., for querying topology data in case of the AFC (c.f., §4.2.1), incurs higher latency, compared to, e.g., direct memory access in case of a single-process integration. Although TCP connection over the local loopback interface is highly optimized in recent Linux kernels, micro-benchmarks [25] executed on our testbed and switch[4] indicate higher throughput and lower latency when using inter-process communication (IPC) mechanisms.

When focusing on latency, LDPEP should be implemented by a fully integrated, monolithic controllet which is however still connected to the message bus in order to leverage the global view and central coordination, as explained for the AFC. Furthermore, we see great performance potential for tighter coupling to the underlying switch hardware. Ideally, local logic would be pushed down to the data plane hardware, which is unrealistic for the case of ASICs, due to their fixed hardware design and restricted access. Network processors (NPUs) or FPGA based switching fabrics offer greater accessibility and flexibility. Most promising considering practicability, but least promising considering performance potential, is an integration into the OpenFlow Agent running on a switch's control plane.

In supporting all 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 them in the following.

6. 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.

6.1 Raw Controller Performance

In our first evaluation, we compared the raw performance of ZSDN with other popular controllers, lacking distribution support.

We used cbench [36] for measuring controller throughput and latency. Cbench emulates switch behavior by sending `OF_PACKET_IN`s (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 controllers to send only one response, `PKT_OUT`, as the result of their processing. For sequential throughput, cbench waits for a response to a sent

[4]http://goo.gl/tOaTKw , http://goo.gl/eHVMnQ

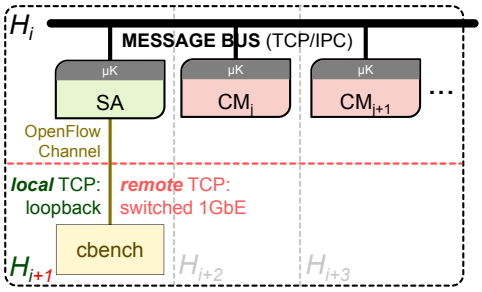


Figure 4. Evaluation setup of control plane (testbed) for raw controller performance evaluation (§6.1) and scalability evaluation (§6.2).

trigger, before sending the subsequent trigger. Thus, sequential throughput depicts inverse controller processing latency. We had 1 switch with 100.000 connected end hosts being emulated by cbench. Cbench and the controllers were run on a testbed consisting of 12 nodes (Intel Xeon E3-1245v2 @ 3.40GHz, 4 physical cores, 16GB RAM) interconnected through a 1Gbps switched Ethernet network.

To investigate the implication of controller locality, we differentiate between cbench, i.e., the switch, and the controller running on the same node (*local*; OpenFlow channel: TCP loopback interface) or on different nodes (*remote*; OF Channel: TCP over switched Ethernet), as illustrated in Figure 4.

Each cbench run is averaging over 60 seconds 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. Here, individual runs on the nodes are completely independent of each other.

We evaluated the following control platforms: (1) ZSDN-TCP/IPC: modular controller framework using single-instance controllets (logical, not physical distribution) with message bus communication using the ZMQ transport mechanism: TCP (loopback) or IPC (UNIX domain sockets); (2) ZSDN-AFC: the Autonomous Forwarding controllet, as introduced in §4.2.1, fully integrated (single process, c.f. §5.3); (3) NOX (verity) [16, 39]: an early academic C++ implementation, popular for its performance; (4) ONOS [27]: Java-based, carrier-grade; (5) Floodlight [5]: Java-based, production-grade; (6) Ryu [34]: Python-based, popular for support of recent OF versions;

Figure 5 shows the **results** of the controller comparison, where error bars depict the standard deviation. Regarding throughput (*local*), NOX performs best with $\approx 369 \pm 1.7$ messages per millisecond (msg/ms). The LDPEP of ZSDN-AFC results in similar figures with $\approx 260 \pm 0.8$ msg/ms. Due to an identified memory inefficiency in the OF library used in ZSDN, we even expect to reach NOX performance.

The performance penalty of distribution shows to be bearable: distributed ZSDN throughput is about 53% of ZSDN-AFC ($\approx 138 \pm 28$ 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 1/3 when using IPC instead of TCP. This contradicts expectations risen through the micro-benchmarks [25], 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

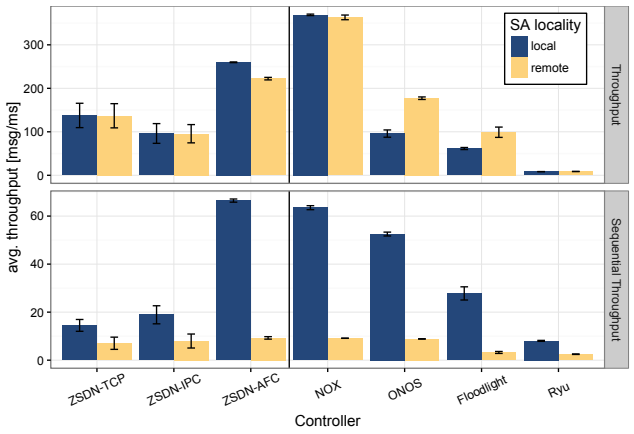


Figure 5. Raw controller performance: comparison of throughput and sequential throughput (inverse latency) for ZSDN and other popular controllers, drilled down by switch-controller locality (*local*/*remote*).

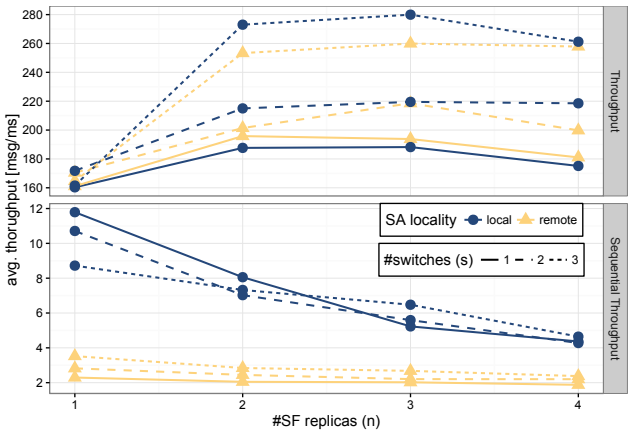


Figure 6. Scalability evaluation (scale-out): modular ZSDN controller with varying number of SimpleForwarding instances and varying number of connecting switches.

slightly better. The Python-based controller Ryu is far off with ≈0.8 msg/ms. Overall, throughput penalties for *remote* OF connection are moderate.

For latency, 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 as discussed in §4.2.5, is bearable.

We are aware that such high throughput would not stem from realistic scenarios considering a small number of switches, however the results provide valuable insights in determining the upper performance bound, e.g., in case of aggregated load from a large number of switches, e.g., in the case of reactive ARP handling.

6.2 Controllet Distribution

To investigate the benefits of distribution, we distribute the most compute-intensive controllets SwitchAdapter (SA) and SimpleForwarding (SF) to dedicated nodes, while running the other controllets on a common node as illustrated in Figure 4. Note, that more compute-intense processing, such as deep packet inspection, would much more benefit from distribution and replication. Furthermore, we replicate the SF with a varying replication factor of n and distribute the instances to dedicated nodes. The SA distributes the total load evenly to these instances (c.f. §5.2.2). For the moment, we keep the replication factor of SAs (k) constant at 1, i.e., using a single SA instance. We additionally vary the number of switches cbench emulates (s). For each switch connection, the SA spawns 4 threads, dedicated to that connection.

The **results** are shown in Figure 6. Even for $n = 1$ (no replication) we achieve 15% higher throughput by placing the SF instance on a dedicated node. However, the single SF instance constitutes a bottleneck: with increasing n, throughput increases, as expected. We do not see a linear increase, since the SA 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 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 $\approx 280 \pm 16$ msg/ms. Note, that when maxing out, results vary much, such that, e.g., throughput seems to drop, as shown in the graph. We repeated the test while scaling-up, i.e., using more powerful nodes (Xeon E5-1650v3 @ 3.50GHz, 6 physical cores), where throughput peaks at $s=5, n=5$ with $\approx 670 \pm 20$ msg/ms.

For $n = 1$, latency increases by about 22% due to the physical separation of SA and SFM and thus one additional hop over the control plane network. In contrast to *remote*, for *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 OF TCP connection over the network instead of the loopback interface is the dominating factor. In more realistic 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 in replicating both, SF and SA, i.e., increasing n as well as k, and keeping n/k balanced, we verified linear scalability, as expected. 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 10GbE or higher on the control plane, counteracting network capacity bottlenecks.

6.3 Performance on White-box Networking Switch Hardware

In our last evaluation, we compare the controller performance on real switch hardware. To this end, we deployed ZSDN-IPC, ZSDN-AFC, and NOX on our 10GbE white-box networking switch, Edge-Core AS5712-54X, featuring an x86 Atom Rangeley CPU with 4 cores at 2.4 GHz, 8 GB RAM (DDR3), and 8 GB Flash memory. Note, that we have added NOX as a baseline. Although not entirely comparable due to differences in the implementation of packet processing of NOX and ZSDN-AFC (both written in C++), code review

of NOX revealed that the implementation is in fact sufficiently similar to allow for a reasonable comparison. The setup is depicted in Figure 7. For the experiments, we were running Pica8 PicOS 2.8 which is running the Open vSwitch OF switch implementation (OVS) with custom drivers to access the underlying hardware switching ASIC. In the data plane, we connect an end host with two 10GbE links to the switch. We use separate network namespaces to isolate the host interfaces, while sharing a common clock. We send packets to the other interface where we measure the throughput of received packets. For measuring latency, we attach sequence numbers to the packets payload, and upon reception on the second interface, reflect them back to the first interface, where we capture both, packet ingress and egress, matching their sequence numbers and determine the round trip latency through their capture timestamps. Here, we send with a rate of 20pps and capture for 240s, resulting in a sample size $N = 4800$ packets.

As for the **throughput results**, we observe throttling due to excessive OVS daemon load: for sending rates $\geq$ 20kpps, we measure receive rates of 8.5kpps for ZSDN and 6.5kpps for NOX, while the OVS daemon consumes 200% CPU, leaving a very limited share for ZSDN/NOX. This is in line with observations we have made earlier: the OVS daemon heavily limits the CP-DP bandwidth. Even when installing flow rules that send traffic to the controller, throughput is limited to about 60Mbps. Modern hardware switches, though OF-capable, are still optimized for traditional DP forwarding, where DP-to-CP traffic is deliberately limited implementing a security mechanism, e.g., to prevent excessive traffic threatening CP operation.

Figure 8 shows the **results of the latency** measurement. For switch-*local* controllers, ZSDN-AFC performs best with $\approx 1080 \pm 112\mu s$ average latency, comparable to NOX with $\approx 1163 \pm 95\mu s$, while the costs for module decoupling over the message bus of ZSDN-IPC result in $\approx 2021 \pm 151\mu s$, clearly showing the preference on the full integration scheme of ZSDN-AFC which still profits fully from centralized view and coordination. For *remote* controllers that run on one of the testbed nodes described above, the TCP connection over the 1GbE network instead of the loopback interface, considerably increases latency by factors of about 2, 1.88 and 1.56 for ZSDN-AFC, NOX and ZSDN-IPC, respectively.

The observed deviation could be identified to stem mostly from the OVS daemon.

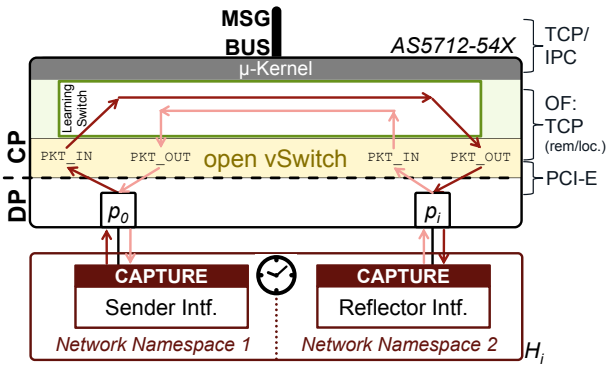


Figure 7. Evaluation setup of data plane for controller performance evaluation on white-box networking switch hardware (§6.3).

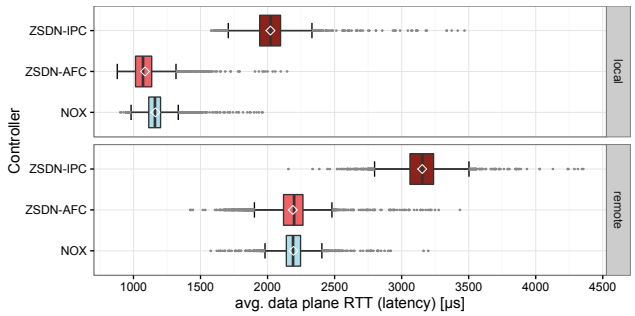


Figure 8. Evaluation on a white-box networking hardware switch: comparison of end-to-end (data plane) latencies (RTT) of modular ZSDN, integrated ZSDN-AFC, and NOX, drilled down by switch-controller locality.

7. ROADMAP TO A HIGHLY SCALABLE HOLISTIC SYSTEM CONTROL PLANE

We would like to discuss how to further improve and leverage the presented concept of event-driven distributed network control.

Our distributed SDN controller is based on content-based filtering of events, in particular, the filtering of data plane events based on header field matching. In larger networks, event notifications might arrive at a high rate, which makes content-based message filtering by the message bus challenging. In our prototype, we used a high-performance topic-based messaging system (ZeroMQ) as a workaround by mapping match fields onto a topic hierarchy. However, such a topic mapping also comes with inherent problems. In particular, the discretization of the event space incurs lack of expressiveness. Also, attributes have to be specified according to the order given by the topic hierarchy. Irrelevant hierarchy levels can be wildcarded, however, efficient wildcard topic matching at high event rates is hard to implement in software.

To solve this problem, we observe that content-based data plane event filtering is conceptually similar to matching header fields during packet forwarding by switches in the data plane. Hardware switches achieve line-rate forwarding performance in the data plane through special hardware (TCAM) supporting also wildcard matching efficiently. So one interesting question is, could we utilize similar hardware for implementing an *SDN message bus appliance* supporting content-based event filtering and routing to subscribing controllets? Similar appliances have already been used in other domains like service-oriented architectures, in speeding-up XML processing [31].

As a second extension, we can further leverage the publish/subscribe paradigm to build a *holistic distributed system controller* not limited to controlling the network elements but to include virtual network functions, end systems (including virtual machines), applications (e.g., client and server processes on the application layer), etc. In other words, we can extend the network control plane to a *holistic system control plane* implemented by a set of distributed controllets, which communicate indirectly through events including not only simple and complex data plane events but any event relevant for controlling and managing the holistic system. As a simple example, consider the migration of a virtual machine (VM), which might also require the migration of virtual

network functions like firewalls, and the adaptation of routes for chaining services. Using event-based communication, we can trigger actions to implement an event-triggered workflow defining the sequence of actions necessary to migrate the VM. For instance, as soon as the VM has been suspended by a VM controllet, an event could be fired that triggers the migration of network functions, which then trigger the adaptation of routes in the network through further events. This way, complex *system management workflows* can be implemented in a decentralized fashion.

8. RELATED WORK

Many early approaches, including Onix [22], 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 [40], 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 [12] is the first SDN approach to allow for local decision making on the switch, however mandating changes of the switching ASIC. Kandoo [17] 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 [13] 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 [23, 2]. 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 [44] 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 [21], 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 [41] 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.

P4 (Programming Protocol-independent Packet Processors) [6] specifies a high-level language for network programming that is not tied to fixed packet header definitions. The P4 compiler maps generic control programs to specific hardware or software platforms of target switches. Thus, P4 is able to fully exploit the capabilities of individual switch hardware, e.g., ASIC, NPU, or FPGA. However, switches in P4 do not take control decisions but merely execute control logic that has been compiled down from a high-level description, i.e., deploying control decisions instead of distributing decision making. Furthermore, P4 does not address the question where its compiler is actually executed, overall showing that control plane distribution is not considered.

9. SUMMARY

In this paper 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 light-weight 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.

We also identified future research directions like hardware-assisted processing of data plane events to further increase the scalability of content-based event filtering in the control plane, or the event-driven execution of system management workflows in a holistic system control plane embracing the network control plane.

10. ACKNOWLEDGMENTS

We would like to thank the students involved in the implementation of the ZeroSDN project, especially Jonas Grunert and Andre Kutzleb. A full list of contributors can be found in [47]. We would also like to thank the autonomous reviewers for their valuable feedback.

11. REFERENCES

[1] 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*, pages 61–66, Sept. 2014.

[2] 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)*, pages 354–359, Nov. 2015.

[3] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch. *SIGCOMM Comput. Commun. Rev.*, 44(2):44–51, Apr. 2014.

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

[5] Big Switch Networks. Floodlight: An Open SDN Controller. http://www.projectfloodlight.org/floodlight/. [Online].

[6] 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.*, 44(3):87–95, July 2014.

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

[8] Broadcom. Github Repository: OF-DPA. https://github.com/Broadcom-Switch/of-dpa. [Online].

[9] 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.

[10] 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*, pages 85–90, Sept. 2015.

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

[12] 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*, SIGCOMM '11, pages 254–265, New York, NY, USA, 2011. ACM.

[13] 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*, ANCS '14, pages 17–28, New York, NY, USA, 2014. ACM.

[14] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[15] Google Inc. Protocol Buffers. https://developers.google.com/protocol-buffers/. [Online].

[16] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.

[17] 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*, HotSDN '12, pages 19–24, New York, NY, USA, 2012. ACM.

[18] Hewlett Packard Enterprise. HP Switch Software OpenFlow v1.3 Administrator Guide nl K/KA/KB/WB 15.18. http://h20566.www2.hpe.com/hpsc/doc/public/display?docId=c04777809, Dec. 2016. [Online].

[19] IEEE 802.1AB: Local and Metropolitan Area Network Standards. Station and MAC-Discovery. *IEEE Std.*, 2005. [Online].

[20] iMatix Corporation. ZeroMQ: Distributed Messaging. http://zeromq.org/. [Online].

[21] 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*, SOSR '15, pages 4:1–4:12, New York, NY, USA, 2015. ACM.

[22] 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*, OSDI '10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[23] 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*, HotSDN '14, pages 133–138, New York, NY, USA, 2014. ACM.

[24] nanomsg Documentation. Differences between nanomsg and ZeroMQ. http://nanomsg.org/documentation-zeromq.html, Jan. 2017.

[25] NetOS group, University of Cambridge Computer Laboratory. `ipc-bench` results database. http://www.cl.cam.ac.uk/netos/projects/ipc-bench, Oct. 2016.

[26] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A Scalable Fault-tolerant Layer 2 Data Center Network Fabric. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 39–50, New York, NY, USA, 2009. ACM.

[27] ON.LAB - ONOS. ONOS - A new carrier-grade SDN network operating system designed for high availability, performance, scale-out. http://onosproject.org/. [Online].

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

[29] OpenDaylight Foundation. OpenDaylight: Open Source SDN Platform. https://www.opendaylight.org/.

[Online].

[30] OSGi Alliance. OSGi: Open Services Gateway initiative. https://www.osgi.org/. [Online].

[31] D. Pasetto, H. Franke, K. Schleupen, D. Maze, H. Penner, H. Achilles, C. Crawford, and M. Purcell. Design and Implementation of a Network Centric Appliance Platform. In *2012 Brazilian Symposium on Computing System Engineering (SBESC)*, pages 204–207, Nov. 2012.

[32] 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. pages 117–130, 2015.

[33] O. C. Project. Github Repository: SAI (Switch Abstraction Interface). https://github.com/opencomputeproject/SAI. [Online].

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

[35] S. Schmid and J. Suomela. Exploiting Locality in Distributed SDN Control. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 121–126, New York, NY, USA, 2013. ACM.

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

[37] 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*, SIGCOMM '15, pages 183–197, New York, NY, USA, 2015. ACM.

[38] D. Thaler and C. Hopps. Multipath Issues in Unicast and Multicast Next-Hop Selection. RFC 2991, RFC Editor, November 2000.

[39] The NOX Controller. Github Repository: NOX Network Control Platform. https://github.com/noxrepo/nox. [Online].

[40] 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*, pages 3–3. USENIX Association, 2010.

[41] 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*, SIGCOMM '15, pages 43–56, New York, NY, USA, 2015. ACM.

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

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

[44] 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.

[45] ZSDN. Github Repository: Java Module Framework. https://github.com/zeroSDN/JMF. [Online].

[46] ZSDN. Github Repository: Zero Module Framework. https://github.com/zeroSDN/ZMF. [Online].

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