The Case for the Software Defined Edge

Matthew P. Grosvenor, Lazaros Koromilas, Hitesh Ballani,
Paolo Costa, Christos Gkantsidis, Thomas Karagiannis

Technical Report
MSR-TR-2014-154

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

Abstract

Despite the hype, the use of Software Defined Networking (SDN) has been limited to simple management tasks. This results both from the narrow interface between the data- and control-plane in today's SDN designs and from the inherent limitations of enforcing policies at switches. We propose SWEDEN, a software-defined edge architecture for enforcing management policies in closed environments like datacenters.

SWEDEN enforces the policies at end hosts with minimal network support. It decomposes administrator specified policy programs into program fragments that (i) require global visibility and hence, run at a centralized controller, (ii) need to be responsive and hence, run at the end host, (iii) need to be on the data path and hence, run at the network interface (NIC). For the last part, SWEDEN leverages programmable NICs and exposes a programmable match-action API, thus allowing delegation of control logic and state to the NIC.

Through examples, we show how such end-based management improves control-plane scalability and responsiveness while allowing for richer policies than previous proposals.

1 Introduction

To manage today's networks, administrators have to manually configure both network devices and end hosts. This is cumbersome and error-prone. To remedy this, software-defined networking (SDN) argues for a simple interface between the data- and control-plane which, in turn, allows switches to be programmed based on high-level policies. OpenFlow [17] is the de facto interface for today's switches, while efforts like Open vSwitch [23] have extended this programming model to end hosts too. However, the narrow data-plane interface restricts a lot of SDN work to simple policies like routing and reachability.

This paper explores an alternate tact for policy-based network management. We observe that while switches have good network layer visibility due to their location, they have limited compute and storage resources. By contrast, *end hosts* (i.e., compute servers, storage servers and middleboxes) have plentiful resources and are aware of high-level semantics for their traffic. Hence, in closed environments like datacenters, diverse management policies can be enforced at end hosts with minimal support from the network.

We propose SWEDEN, a software-defined edge architecture for policy-driven programming of end hosts. SWEDEN comprises three components: a logically centralized controller, end host control modules and programmable network interfaces (NICs). Of these, only the NICs are on the data path. This design has two key features. First, since end hosts can run actual control programs, it leads to a natural distribution of the control-plane between the controller and the ends. Second, the NICs expose a programmable match-action API with both programmable matches and actions. This API is the interface between SWEDEN's data- and control-plane, and means that we can delegate requisite control logic and state to the NICs themselves.

SWEDEN can enforce management policies and even implement new data-plane functionality. The administrator expresses policies using event-driven programs, written in a high-level language, that are to be run at an *idealized controller*. This policy program is automatically decomposed into controller, end-host and NIC programs. Such decomposition leverages the strengths of each layer– logic requiring global visibility resides at the controller, logic that needs to be responsive or requires application-level semantics resides at the end host module, and only minimal logic that is necessary on the data path is on the NIC itself. Through a few example policies that are either hard or impossible to implement using SDN, we illustrate the benefits of such an approach: *better scalability* since each end is only responsible for traffic it observes, greater *control-plane responsiveness* as control logic resides at the end host itself, a *performant data-plane* since only necessary computation and state resides on the data path, and the ability to enforce *richer policies* that are stateful or require application-level semantics. We also describe our implementation efforts and a case study showing that our prototype can implement a simple per-packet load balancing policy at line rate.

The observation regarding the benefits of end host based network management is not new [7,11,16,23,27,28]. However, even with these proposals, the interface to the end host data plane is still a variant of OpenFlow. Instead, SWEDEN embodies an extreme design point with a distributed control-plane and a programmable data-plane that can be programmed to do on-path stateful computations.

2 Design

In this section, we provide an overview of the SWEDEN architecture and programming abstractions and then describe the implementation of a few example policies. SWEDEN targets environments in which (some components of) end hosts are owned by a single administration domain and can therefore be trusted. In this paper, we focus on datacenters, although our approach can also be applied in other scenarios such as enterprise or home networks [11,12].

2.1 Design Overview

The design of SWEDEN has been driven by two main goals. The first is to have a *distributed control plane*, which leverages the global knowledge available at the controller as well as the large amount of resources and application-level visibility available at the end hosts.

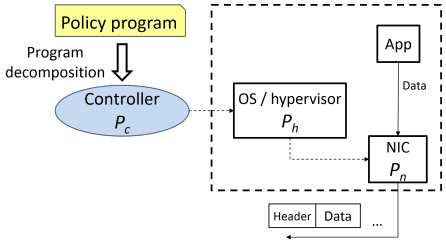

Figure 1: The Sweden architecture. Dashed lines represent control-plane operations while solid lines indicate data-plane ones.

This is achieved by splitting the control logic between a centralized controller and a set of control modules running within the OS or hypervisor at the end hosts.

Second, we want to provide a *programmable data plane*, which can implement richer policies than the ones currently supported by SDN. We leverage recent availability of programmable NICs that allow actual programs to be executed on the NIC. We exploit this capability in Sweden by delegating the policy logic that is necessary on the data path to the NICs. This ensures high performance while still retaining a high degree of flexibility.

The Sweden architecture is depicted in Figure 1. Policies are specified using a *policy program*, written in a high-level language, which is conceptually able to observe all packets sent and received by end hosts in the network. This allows administrators to focus on the algorithms underlying the policies to be enforced instead of the details of their implementation. We describe the programming model used to express the policies in Section 2.2.1.

A policy program p is decomposed into program fragments to be run at the controller (p_c), at each end host (p_e) and within the NIC (p_n). Conceptually, $p = p_c + \cup_{e \in E}\ p_e + \cup_{n \in N}\ p_n$, where E and N are the set of end hosts and NICs governed by the policy. Here, "=" implies extensional equality — given the same inputs, the programs will generate the same output, i.e. return values and global state. Like the original policy program, the controller program and the host program are implemented in a high-level language. For efficiency reasons the NIC program, instead, uses the restricted set of API described in Section 2.2.2. Unlike traditional SDN proposals, since the control logic is implemented in the centralized controller, Sweden requires only limited functionality from the switches, which we describe in Section 2.2.3.

In §2.4 we show how we can leverage existing work to automatically partition policy programs into program fragments. In the following section, we rely on a few manually decomposed examples to illustrate the benefits of our design.

Match	—>	Action
<match-exp>	—>	f (pkt, [state parameters])

Table 1: Match-action rules in the NIC

2.2 Programming Abstractions

Next, we briefly describe the programming model used by administrators to express their policies in Sweden and the API that we use to implement the NIC control modules. We then describe the switch functionality required.

2.2.1 Policy programming model

We opted for an event-based programming model to express policy programs. This reduces the complexity for developers (as they only need to worry about handling a small set of events) and simplifies the decomposition process (as the program structure is fixed). We support a pre-defined set of objects, namely `packet`, which allows developers to easily access packet fields, `topology`, which comprises the link and nodes (end host, switches, and middleboxes) in the datacenters, and `timer`, which enables timeouts or deferred actions. Developers can also define their own data structures by composing standard types such as `list` and `hashTable`. Instances of these custom data structures (hereafter called *state*), however, must be explicitly listed in the program preamble and cannot be dynamically allocated as doing this would complicate code static analysis.

We identified four classes of events and defined the following methods to handle them:

- `onPcktRcvdFromApp`: invoked when a packet is received from the application;
- `onPcktRcvdFromNet`: invoked when a packet is received from the network;
- `onTopologyChange`: invoked when there is a change in the topology, e.g., due to a link or server failure;
- `onStateChange`: invoked when one of the state object changes, e.g., a new tenant is admitted into the network.

A timeout can be created using the primitive `createTimer`, passing as parameters the time at which it should fire and the callback function to be invoked. We provide standard primitives to handle packets such as `forward`, `drop`, and `duplicate`. Beside common fields such as `src` and `dst`, the `packet` object also contains some meta-fields such as `priority` and `rateLimiter` that are not mapped to header fields but are used to indicate how the packet should be queued.

Section 2.3 provides a few examples of implementing policies using this programming model in pseudo-code.

2.2.2 NIC API

The challenge with allowing NIC programmability is to achieve flexibility without sacrificing performance. To balance this, we expose a restricted NIC programming

model. A NIC program comprises a set of tables containing match-action rules, as shown in Table 1. The match operation is specified as a matching expression operating on the packet header. For the action, instead of a pre-defined set of actions, we allow *action functions*. An action function is a loop-free function that takes as input a packet and returns the set of next hops the packet should be forwarded to. Here a next hop can be the host, a NIC port, or even another table. The function can have other optional state parameters that carry shared state. The function can manipulate the packet (header and body), the state parameters, and create, delete, modify the match-action rules in the NIC.

Match-action programming is a natural fit for two reasons. First, it is possible to efficiently implement lookup tables in hardware using TCAMs. Secondly, thanks to the popularity from the OpenFlow community, match-action programming is already a familiar model for programmers.

2.2.3 Network support

SWEDEN's end-based approach to network management requires that end hosts be able to control how their packets are routed through the network. Recent proposals have shown how existing technologies like MPLS and VLANs can be used to achieve source routing in datacenters [7,18]. Here, end hosts specify the path of a network packet as a label in the packet header, e.g., through an MPLS label or VLAN tag, and switches perform label-based forwarding.

Such source routing requires the controller to configure the label forwarding tables at switches. For MPLS, this can be achieved through a distributed control protocol like LDP [5] while for VLANs, Spain [18] describes a solution involving multiple spanning trees. Hence, label based forwarding and the corresponding control protocol is the primary functionality SWEDEN requires of the underlying network. Additionally, most existing switches already support statistics gathering capabilities (e.g SNMP) and priority-based queuing. These can be exploited by SWEDEN to implement richer policies.

2.3 Example policies

We now use four example policies to illustrate SWEDEN's operation. We chose these examples as they are either difficult or impossible to implement using SDN. The first three policies can be implemented with an idealized form of SDN (that is not restricted to the OpenFlow protocol) but in an approximate form and with higher latency for new flows and much greater control overhead. The last policy involves new data-plane functionality that requires application semantics and cannot be implemented using SDN.

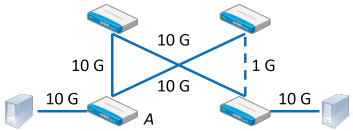

Figure 2: Example of asymmetric topology.

2.3.1 Weighted Load Balancing

Datacenters today use ECMP to load balance traffic among multiple equal cost paths. However, ECMP assumes that the underlying topology is balanced and regular which is often not true in practice [34]. For example, in Figure 2, ECMP would distribute the traffic at switch A evenly across the two upstream links even though subsequent links along the paths have very different capacities. This can result in unfairness and poor tail performance. Consequently, Zhou et al. [34] propose WCMP, a weighted version of ECMP that balances traffic across links in a weighted fashion. In the example in Figure 2, WCMP can be used to balance traffic across the two upstream links of A in a 10:1 ratio.

```
Require: Datacenter topology– topology

   //Global program state
 1: cachedPaths:[src, dst] -> {[Path1, Weight1], ...}

   //Helper functions
   //choose a path in a weighted fashion
 2: path weighted_select({[path, weight], ...})
   //given a src and dst, generate a weighted set of paths
 3: {[path, weight], ...} gen_paths(src, dst)

   //Event-driven functions
 4: void onTopologyChange (){
 5: for Link l: topology.getLinks() do
 6:    l.weight = compute_weight()
 7: }

 8: void onPcktRcvdFromApp (packet p) {
 9: if [p.src, p.dst] not in cachedPaths then
10:    cachedPaths[p.src, p.dst] = gen_paths(p.src,
       p.dst)
11: p.path = weighted_select(cachedPaths[p.src,
    p.dst])
12: send(p)
13: }
```

Figure 3: Policy program implementing weighted load balancing.

Policy program. We consider the following high-level policy– "Use *per-packet* WCMP to balance the traffic across all links". Figure 3 shows a policy program that implements this policy. Note that this an idealized policy program that is invoked, without any delay, every time an end host sends or receives a packet. The key program logic resides in two event functions: `onTopologyChange` function that is invoked whenever the network topology changes and the `onPcktRcvd-FromApp` function that is invoked when a host sends a

packet. At a high level, the policy program performs three operations– (i). It uses knowledge of the datacenter topology, represented by as a graph, to associate a weight with each link in the graph (`onTopologyChange` function). (ii) For each flow, it determines the weight for each path to the destination. A path's weight is the probability with which it should be used and is a product of the normalized weights of all links along the path. This step needs to be responsive to avoid delaying flow initiation, but the results can be cached for subsequent packets of the flow (lines 9–10). (iii). For each packet, it chooses a path in a weighted fashion (line 11).

Program fragments. With Sweden, we can decompose this program into program fragments to be run at the controller, end host and the data path (i.e. NIC). The controller program performs the first operation mentioned above; it implements the `onTopologyChange` function, and informs end hosts of the new topology and per-edge weights. The host program performs the second operation by implementing the `gen_paths` function. The NIC program, shown in Figure 4, performs the third operation.

```
Require: End host implements gen_paths

   //Intital Match-Action Table
 1: <src:*, dst:*, sport:*, dport:*> -> f(p)

   //Helper functions
   //Choose a path in a weighted fashion
 2: path weighted_select({[path, weight], ...})
   //Given a path, generate its routing label
 3: label generate_label_for_path(path)
   //Given routing label, determine the next hop
 4: next-hop next_hop(label)

 5: next-hops f(Pkt p) {
 6: weightedPaths = HOST.gen_paths(p.dst)
 7: create_rule(<p.src, p.dst, p.sport. p.dport> ->
    g(pkt, weightedPaths))
 8: return g(p, weightedPaths)
 9: }

10: next-hops g (Pkt p, {[path, weight]} paths) {
11: path = weighted_select(paths)
12: p.label = generate_label_for_path(path)
13: return next_hop(p.label)
14: }
```

Figure 4: NIC program for weighted load balancing.

We detail the operation of the NIC program. It comprises a default rule that matches all packets and applies function `f` to the packet (line 1). When a new flow starts, this function is invoked. The function uses an RPC to the host to obtain the set of paths to the destination and their weights (line 6). It then inserts a match-action rule such that subsequent packets of the flows are processed by function `g` (line 7). This match-action rule is ephemeral and times out when the flow finishes. The state parameter for the function `g` (i.e., `paths`) contains the flow's paths and their weights, and it selects the packet's path in a weighted fashion .

We note that in this example, there is a straightforward mapping between functions in the original policy program and the program fragments. However, in general, the decomposition could be more involved.

2.3.2 Rate control

Proposals for congestion control in datacenters can be roughly divided in three different classes. The first one contains traditional congestion control algorithms like TCP and its variants [2,3,30], which operate in a distributed fashion. At the other end of the spectrum, we find recent proposals for centralized congestion control that rely on global knowledge and centrally compute the rate for each flow [22]. Finally, there are hybrid solutions that leverage coarse-grained network information, e.g., link congestion or utilization, to guide fine-grained rate allocation at the edge, e.g. [12,19].

Here, we focus on the last class and show how we can implement in Sweden a simplified version of the hybrid solution described in [12]. The solution consists of two phases. In the first one, the centralized controller computes a price p, $0 \leq p \leq 1$, for each network link based on its utilization (details in [12]). Next, link prices are used to estimate a flow's fair sending rate. A flow's fair share across a link A is estimated through following feedback control loop:

$$X_A(t+1) \leftarrow X_A(t) + kX_A(t)[1 - p_A(t) - p_A(t)u_A(t)], \quad (1)$$

where X_A is the nominal fair share in link A at time t, k is a gain parameter, and $u_A(t)$ is the utilization of the resource A at time t. The sending rate of a flow is the minimum of its share across all links it traverses.

Policy program. The policy program for this solution is reported in Figure 5. The key program logic resides in two functions that are invoked periodically. The first function, `compute_prices`, is invoked at coarse timescales (seconds) and uses link utilization information to compute link prices. The second function, `update_rates`, is invoked at fine timescales (milliseconds) and uses link prices to update the rates for individual flows.

Program fragments. Sweden can decompose the program as follows. The controller runs the `compute_prices` function by periodically polling switches for statistics on link utilization. The host program extracts all resource information that is relevant for its local flows and periodically recompute the rates, which are then enforced by the NIC.

2.3.3 Anycast

The example in Section 2.3.1 focused on load balancing of traffic across multiple paths to a destination. Here, instead, we consider the complementary policy of anycasting—load balancing among different destinations. This is typical of web or IO traffic in which requests are usually dispatched to any of the available web

```
Require: Datacenter topology– topology

   //Global program state
 1: activeFlows: {Flow} //the set of active flows

   //Helper functions
 2: compute_prices(topology)
 3: enforce_rate(flow, rate)
 4: update_rate(flow) //computes the flow rate using Eq. 1

   //Event-driven functions
 5: void compute_prices (){
 6: for Link l: topology.getLinks() do
 7:    l.price = compute_price()
 8: createTimer(1s, compute_prices)
 9: }

10: void update_rates (){
11: for Flow f: localhost.getActiveFlows() do
12:    f.rate = update_rate(f)
13:    enforce_rate(f, f.rate)
14: createTimer(1ms, update_rates)
15: }

16: void onPktSend (Pkt p) {
17: if flow not in activeFlows then
18:    update_rate(flow)
19: send(p)
20: }
```

Figure 5: Policy program implementing centralized congestion control.

servers or storage replicas. Applications interact with the service using a virtual IP address (VIP), which is then remapped to the direct IP address (DIP) of the selected destination. This can be achieved using custom hardware appliances or software load balancers [21].

Conceptually, the policy can be expressed as "given a request (e.g., HTTP `GET` or IO `read`), forward it to one of the available destinations in a (weighted) round-robin fashion". Due to space constraints, we do not show the policy program. Instead, we explain the operation of various program fragments. The controller leverages global visibility to keep track of the list of available destinations for each virtual IP and when a change is detected (e.g., due to a server failure), it immediately notifies the end hosts. When a new flow starts, the NIC contacts the host's control module to retrieve the destination to use. Next, for each packet belonging to the flow, the NIC performs a NAT operation, i.e. it replaces the virtual IP used by the application with the dynamic IP returned by the host. This process is completely transparent to applications, which only see the virtual IP address.

While such anycasting is used prominently by distributed datacenter applications today, Sweden can also be used to implement enhanced versions that take into account application-specific and/or dynamic criteria to select the destination. For instance, we could select as destination the server with the lowest number of outstanding requests in its queue or with the lowest CPU load. Like in the example above, the controller provides the end hosts with a list of available destinations, which are then passed to the NIC. When a new flow is created, the NIC sends a request to each of them, asking for the current value of the desired metric (e.g., the CPU load). The receiving NIC issues an RPC to the local host to obtain the requested measure and sends it back to the source NIC. Once all replies have been received (or the timeout has expired), the source NIC selects the best destination and establishes a connection to it. Finally, as described above, the NIC takes care of rewriting the packet destination address when sourcing a new packet.

2.3.4 *Message-awareness*

We conclude our list of examples by briefly discussing a novel policy that takes into account application-level semantics in network processing.

Traditional network policies operate either at the flow or packet granularity. Often, however, neither choice is fully satisfactory. For example, balancing traffic across multiple paths on a per-packet basis achieves the best load distribution but it can negatively impact application throughput due to high packet reordering. Conversely, load balancing on a per-flow basis ensures that packets are received in order but it can lead to severe load unbalance in presence of flows with heterogeneous sizes [1].

We argue, instead, that in many cases a better strategy is to partition packets based on application-level message boundaries. These can be detected either programmatically by the end host (e.g., using a custom API or relying on the socket `send()` primitive) or inferred at the NIC by inspecting the content of the socket buffer. Once the message boundaries have been identified, the NIC can then enforce the specific policy when sending a packet. For example, a message-aware traffic policy could force all packets belonging to the same message to be routed along the same path. Similarly, the NIC could assign priorities to packets based on the kind of message (e.g., data vs. control message) they belong to, rather than statically assigning priorities based on packet headers.

2.4 Program decomposition

Automatic program decomposition has been a very active research area over the past decade; for web applications [9,32,33], sensor networks [20], mobile applications [6,10,24], databases [8], and COM applications [15]. Some of these techniques impose a structure and constraints on the original program to make it amenable to decomposition [15,20,24,32], some are restricted to decomposition at function boundaries [6,10], while some support general programming models [8,9]. Merlin [28] also uses program decomposition for simple reachability and QoS policies. Instead, we support richer policies and offload

general computations to the data plane.

We leverage these ideas to automatically decompose Sweden's policy programs. The basic idea is to perform control flow analysis to identify the basic blocks of each function. Blocks that use global state are executed in the centralized controller. For the rest, we decide their execution as follows. We classify blocks that either use instructions that cannot execute on the NIC or install flow rules as being on the "slow path" (for the latter we assume that they will be executed infrequently); otherwise, blocks that forward or process packets belong to the "fast path"; the rest are assigned as their parent block. We aim to execute slow path blocks in the end host control module and fast path blocks in the NIC. We decompose functions that contain both fast and slow blocks by using a RPC between the NIC and the host. During such RPCs, we also need to copy the necessary state to perform the computation. In our examples, identifying that state has been easy, but, in general, this is a challenging problem. More broadly, we are investigating techniques that use domain-specific information to identify the best place to run the user programs depending on the capabilities (and execution cost) of the NICs, hosts, and controller.

3 Implementation

Our testbed consists of four Intel Xeon E5-1620 machines running at 3.70GHz with 16GB of installed memory. Each machine is equipped with a dual-port 10GbE Netronome NFE-3240 programmable flow processor NIC [35]. The flow processor NICs include a 40 core, 8 way multithreaded CPU (320 threads), and 4GB of memory. The CPU can perform CAM operations on main memory. The cards are programmed in a variant of C89, with explicit memory management for DRAM, scratch memory and general purpose registers. The network switch is a 40x10Gbps Blade RackSwitch.

We are extending the NIC firmware to be able to execute control programs and keep state in the NIC. We use a fast CAM assisted hashtable to implement a match-action flowtable. In addition, we have implemented a lightweight Remote Procedure Call (RPC) mechanism so that the cards can call functions on the host. For example, the card may need to request assistance from the host in the case of a flow table lookup failure. Our action functions are currently written in C and run directly on the card hardware. In future, we envisage a micro-interpreter would be used to provide a safe execution environment for action functions.

4 Weighted load balancing: A case study

As a proof-of-concept, we experimented with the weighted load balancing policy (§2.3.1). Our testing configuration emulates the arrangement shown in Figure 2. To do so, we use two hosts. The primary interfaces on each NIC are connected together at 10Gbps. The secondary interfaces are connected together at 1Gbps. This creates two paths with different capacities between the two machines. The NICs can "source route" between these paths by choosing which port to forward traffic over. A single network interface is exposed at the OS level so that applications are unaware of the multiple paths in the network.

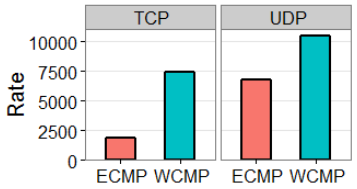

Figure 6: Aggregate throughput for the ECMP and WCMP scenario.

The NICs run our custom firmware and implement the NIC fragment of the WCMP policy—which controls how packets are balanced between the two physical ports. We use per-packet balancing, instead of per-flow balancing to highlight Sweden's ability to apply per-packet policies at high rate. In the default case, we use the same weights for both paths, thus implementing per-packet ECMP. When packets arrive at a NIC, they are equally distributed between both ports. Figure 6 shows the performance in this case (confidence intervals are within 1% of the mean values shown). For TCP, the throughput is dominated by the capacity of the slowest link peaking at 2Gb/s as we'd expect. For the UDP we get a throughput of about 7Gb/s. NIC driver issues mean that each host is limited to 12Gb/s. When divided equally between the two paths, we get 6Gb/s through the 10GbE path plus 1Gb/s through the GbE path.

Next, we apply the weighted select function (see Figure 3) to enforce a 10:1 ratio between ports. Figure 6 shows the results. With UDP, we achieve close to the full 11Gb/s which is our topology's min cut. The TCP performance is lower due to in-network reordering of packets [13]. Modifying TCP's congestion control algorithm can alleviate such issues [25]; however, the goal of our experiment is to highlight how Sweden can implement this simple policy and benefit unmodified applications running vanilla TCP.

5 Discussion & concluding remarks

Sweden decentralizes the control plane, by allowing end hosts to actively participate in policy implementation. At the same time, Sweden envisions a wider data-plane interface through programmable NICs.

Our existing implementation, though at its early stages, highlights that such as shift is feasible and can unlock capabilities that are currently absent in SDNs by i) extending the programmability of the data-plane through

programmable actions, ii) overcoming a number of inherent switch limitations such as the limited computation capabilities (e.g., compute power, memory), the absence of visibility on application semantics and the need to enforce policies for large number of traffic flows at increasingly higher and higher speeds. Instead, operating at the edge provides application context and high computation capabilities that can be devoted to a smaller fraction of the traffic compared to switches.

On the other hand, we recognize that some recent proposals leverage flow coordination capabilities only found within (non-standard) switches [4,14,31]. We envision that Sweden could support these schemes using the same decomposition techniques we discussed, with an additional policy program that runs directly at the switches.

As a final note, we stress that operating at the edge can easily extend the set of management policies with properties that are elusive or hard to guarantee in today's SDNs; examples here include per-flow consistency during updates [26], application-aware traffic engineering or message level routing (see previous section). Taking this a step further, we believe that Sweden's architecture opens up the capability of a cross-resource management plane by extending programmable control to resources beyond the network (e.g., storage [29]).

6 References

[1] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI* (2010).

[2] Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M. Data Center TCP (DCTCP). In *SIGCOMM* (2010).

[3] Alizadeh, M., Kabbani, A., Edsall, T., Prabhakar, B., Vahdat, A., and Yasuda, M. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *NSDI* (2012).

[4] Alizadeh, M., Yang, S., Sharif, M., Katti, S., McKeown, N., Prabhakar, B., and Shenker, S. pFabric: Minimal Near-optimal Datacenter Transport. In *SIGCOMM* (2013).

[5] Andersson, L., Doolan, P., Feldman, N., Fredette, A., and Thomas, B. LDP Specification, 2001. RFC 3036.

[6] Balan, R. K., Satyanarayanan, M., Park, S. Y., and Okoshi, T. Tactics-based remote execution for mobile computing. In *MobiSys* (2003).

[7] Casado, M., Koponen, T., Shenker, S., and Tootoonchian, A. Fabric: A Retrospective on Evolving SDN. In *HotSDN* (2012).

[8] Cheung, A., Madden, S., Arden, O., and Myers, A. C. Automatic partitioning of database applications. *Proc. VLDB Endow. 5*, 11 (July 2012).

[9] Chong, S., Liu, J., Myers, A. C., Qi, X., Vikram, K., Zheng, L., and Zheng, X. Secure web applications via automatic partitioning. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 31–44.

[10] Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., and Patti, A. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems* (New York, NY, USA, 2011), EuroSys '11, ACM.

[11] Dixon, C., Uppal, H., Brajkovic, V., Brandon, D., Anderson, T., and Krishnamurthy, A. Ettm: A scalable fault tolerant network manager. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 7–7.

[12] Gkantsidis, C., Karagiannis, T., Key, P., Radunovic, B., Raftopoulos, E., and Manjunath, D. Traffic Management and Resource Allocation in Small Wired/Wireless Networks. In *CoNEXT* (2009).

[13] Han, H., Shakkottai, S., Hollot, C. V., Srikant, R., and Towsley, D. Multi-path tcp: A joint congestion control and routing scheme to exploit path diversity in the internet. *IEEE/ACM Trans. Netw. 14*, 6 (Dec. 2006), 1260–1271.

[14] Hong, C.-Y., Caesar, M., and Godfrey, P. B. Finishing Flows Quickly with Preemptive Scheduling. In *SIGCOMM* (2012).

[15] Hunt, G. C., and Scott, M. L. The coign automatic distributed partitioning system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 1999), OSDI '99, USENIX Association.

[16] Karagiannis, T., Mortier, R., and Rowstron, A. Network exception handlers: Host-network control in enterprise networks. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication* (New York, NY, USA, 2008), SIGCOMM '08, ACM, pp. 123–134.

[17] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: Enabling Innovation in Campus Networks. *CCR* (2008).

[18] Mudigonda, J., Yalagandula, P., Al-Fares, M., and Mogul, J. C. SPAIN: COTS Data-center Ethernet for Multipathing over Arbitrary Topologies. In *NSDI* (2010).

[19] Munir, A., Baig, G., Irteza, S. M., Qazi, I. A., Liu, A., and Dogar, F. Friends, not Foes — Synthesizing Existing Transport Strategies for Data Center Networks. In *SIGCOMM* (2014).

[20] Newton, R., Toledo, S., Girod, L., Balakrishnan, H., and Madden, S. Wishbone: Profile-based partitioning for sensornet applications. In *Proc. of NSDI* (Berkeley, CA, USA, 2009), USENIX Association, pp. 395–408.

[21] Patel, P., Bansal, D., Yuan, L., Murthy, A., Greenberg, A., Maltz, D. A., Kern, R., Kumar, H., Zikos, M., Wu, H., Kim, C., and Karri, N. Ananta: Cloud Scale Load Balancing. In *SIGCOMM* (2013).

[22] Perry, J., Ousterhout, A., Balakrishnan, H., Shah, D., and Fugal, H. Fastpass: A Centralized "Zero-Queue" Datacenter Network. In *SIGCOMM* (2014).

[23] Pfaff, B., Pettit, J., Koponen, T., Amidon, K., Casado, M., and Shenker, S. Extending Networking into the Virtualization Layer. In *In: 8th ACM Workshop on Hot Topics inNetworks (HotNets-VIII).New YorkCity,NY(October 2009.*

[24] Ra, M.-R., Sheth, A., Mummert, L., Pillai, P., Wetherall, D., and Govindan, R. Odessa: Enabling interactive perception applications on mobile devices. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2011), MobiSys '11, ACM, pp. 43–56.

[25] Raiciu, C., Paasch, C., Barre, S., Ford, A., Honda, M., Duchene, F., Bonaventure, O., and Handley, M. How hard can it be? designing and implementing a deployable multipath tcp. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 29–29.

[26] Reitblatt, M., Foster, N., Rexford, J., Schlesinger, C., and Walker, D. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 Conference on*

Applications, Technologies, Architectures, and Protocols for Computer Communication (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 323–334.

[27] Shieh, A., Kandula, S., and Sirer, E. G. SideCar: Building Programmable Datacenter Networks without Programmable Switches. In *HotNets* (2010).

[28] Soulé, R., Basu, S., Kleinberg, R., Sirer, E. G., and Foster, N. Managing the network with merlin. HotNets.

[29] Thereska, E., Ballani, H., O'Shea, G., Karagiannis, T., Rowstron, A., Talpey, T., Black, R., and Zhu, T. IOFlow: A Software-defined Storage Architecture. In *SOSP* (2013).

[30] Vamanan, B., Hasan, J., and Vijaykumar, T. Deadline-aware datacenter tcp (d2tcp). In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 115–126.

[31] Wilson, C., Ballani, H., Karagiannis, T., and Rowtron, A. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *SIGCOMM* (2011).

[32] Yang, F., Shanmugasundaram, J., Riedewald, M., and Gehrke, J. Hilda: A high-level language for data-driven web applications. In *Proceedings of the 22Nd International Conference on Data Engineering* (2006), ICDE '06.

[33] Zdancewic, S., Zheng, L., Nystrom, N., and Myers, A. C. Secure program partitioning. *ACM Trans. Comput. Syst. 20*, 3 (Aug. 2002), 283–328.

[34] Zhou, J., Tewari, M., Zhu, M., Kabbani, A., Poutievski, L., Singh, A., and Vahdat, A. WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers. In *EuroSys* (2014).

[35] Netronome flownics. `http://netronome.com/product/flownics/`.