Leveraging In-Network Application Awareness

Nik Sultana
University of Pennsylvania
Philadelphia, USA

ABSTRACT

This paper describes a novel approach to Network-Application Integration that leverages ideas from the integration between the network and in-network programs written in languages such as P4. That integration relies on models of the resources that a network makes available on the one hand, and the needs of in-network programs on the other. This paper extends that integration to also include applications' needs on top of in-network programs.

This approach thus integrates the network, in-network programs, and applications at the edge of the network, including clients and servers. We explore the properties of this integration, analyse the information on which such an integration can rely, describe a prototype implementation of this idea, and apply it to a network scenario that involves heterogeneous hardware and different applications.

CCS CONCEPTS

• **Networks** → **Programmable networks**; • **Computing methodologies** → **Distributed computing methodologies**.

KEYWORDS

software-defined networking, dataplane disaggregation, distributed systems, network scheduling

ACM Reference Format:
Nik Sultana. 2021. Leveraging In-Network Application Awareness. In *Proceedings of ACM SIGCOMM 2021 Workshop on Network-Application Integration (NAI'21).* ACM, New York, NY, USA, 5 pages.

1 INTRODUCTION

Applications and networking are usually separated by abstractions that decouple them. This separation helps simplify implementations on either side of the abstraction. But this same separation can stymie finer-grained coordination between applications and the network.

Integrating applications and networking can enable better coordination and resource usage. One approach for integration involves weakening the abstractions that separate applications and networking. While this can benefit performance [7], it requires significant engineering effort and this has knock-on effects on the effort required to maintain and reuse software.

This paper describes an approach that retains the existing decoupling between applications and networking but integrates the two by using models about application *needs* and network *resources*. It does not require customisation of software or hardware.

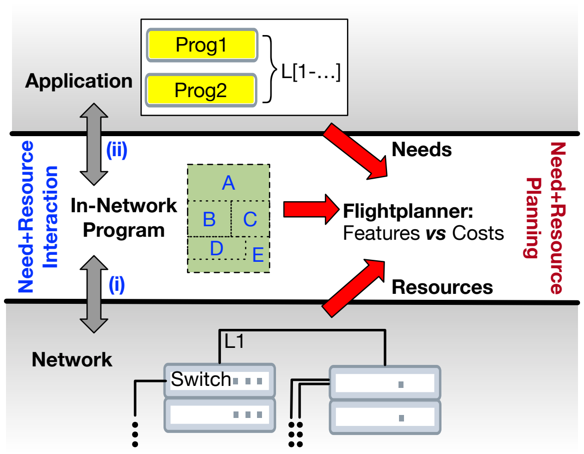

Figure 1: We highlight two levels of integration: (i) Flightplan (§3) uses network awareness for in-network programs, splitting an in-network program into five subprograms A-E and placing them on suitable hardware; (ii) This paper proposes an extension (§4) to integrate applications too.

This approach builds on Flightplan [13], a system designed to gain visibility across abstraction layers in Software-Defined Networking (SDN) to take advantage of the increased availability of heterogeneous programmable hardware in networking, including programmable switches and smart NICs.

Flightplan's approach involves reasoning about in-network programs and network resources—including topology, devices, and their resources. This information is reduced to rules that are processed by *Flightplanner*, an open-source reasoning engine that forms a core component in Flightplan.[1]

Fig. 1 sketches how to extend Flightplan to integrate three domains—the network, in-network programs, and *applications*. The paper describes a prototype implementation of such an *application-aware* Flightplan, sketched in Fig. 2. It adapts Flightplan's reasoning-based approach to obtain application awareness. This initial study suggests that this approach can improve network-application coordination and ease of configuration, and forms the foundation for future research on both the performance and correctness of configuration changes.

This work highlights key research challenges for this approach to integration. These challenges include: (i) gathering comprehensive application needs and modelling their changes (extending §2.1), (ii) how to symbiotically integrate applications with in-network programs (discussed further in §4), (iii) how to mirror this integration inside Flightplanner's core reasoning engine to gain better

[1]flightplan.cis.upenn.edu

performance than the current prototype in which changes are only being done at the periphery.

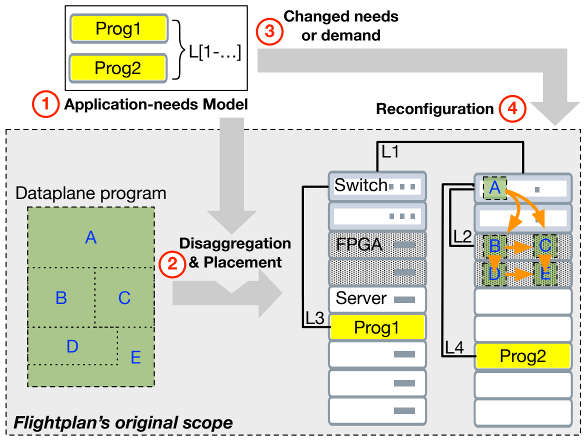
Prog1
Prog2
L[1-...]
① Application-needs Model
③ Changed needs or demand
Reconfiguration ④
Dataplane program
A
B
C
D
E
Disaggregation ② & Placement
Switch
FPGA
Server
Prog1
Prog2
L1
L2
L3
L4
Flightplan's original scope

Figure 2: Overview of *Application-aware Flightplan.* ① Information about applications using the network is included in a model that is made available to Flightplanner, which is extended as described in this paper. ② Flightplanner is run as part of Flightplan. The extended Flightplanner allocates resources in a way that also accommodates application needs. ③ Application needs might change as a result of upgrades, maintenance, failures, external demand. ④ Flightplanner recomputes its allocation model to account for these changes.

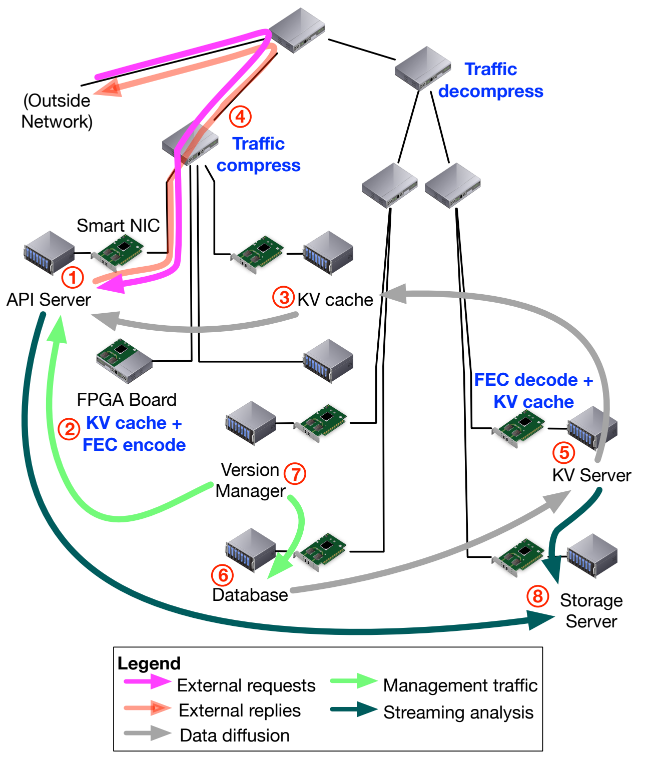

Figure 3: Example scenario for leveraging in-network application awareness. It is explained further in §2.

2 USE-CASE

This section describes an example scenario that features commonly-occurring applications, equipment, and traffic patterns [2, 10]. It also involves a higher degree of mutual awareness of application *needs* and network *resources*. We will see how this awareness can be put to use to benefit the system.

Fig. 3 sketches a small subset of a datacenter network servicing external requests. The network includes a range of hardware. The hardware details and network topology are usually unknown to applications.

In this use-case, API requests are made over a version of HTTP. ① Requests reach servers from the Internet. Servers analyse requests and in turn may produce requests to internal systems, such as Key-Value (KV) stores, in order to service the external request. We call these *sub-requests*. ② In our scenario, sub-requests benefit from in-network programs to improve performance and reliability. These programs are analysed and placed using Flightplan (§3). Examples of such programs include: (i) FEC above the physical layer and (ii) low-latency KV caching to reduce pressure on the KV servers and reduce overall service latency. ③ A host-based, mid-latency KV cache is placed behind the first cache. KV look-ups are a staple of modern datacenter networks and often cascaded [4]. ④ In our example, east-west traffic is compressed using inline logic to reduce network utilisation of the core. This logic is placed by Flightplan based on the available resources. ⑤ KV requests can ultimately reach a server. Assisting logic, such as a last caching layer, can be placed on nearby programmable resources such as the server's smart NIC. Such NICs have seen increased research and deployment over the last years [5, 6, 8, 9]. Application-awareness enables us to better utilise such resources. ⑥ KV servers are populated by batch analysis tasks carried out on structured data stores. ⑦ A management suite of software oversees nodes' health, and may update or roll-back the software used in each node. ⑧ Logs and telemetry are streamed for analysis to determine service usage and trends.

2.1 Application needs

The needs of different applications on the network are reflected by their traffic patterns: their timing, bandwidth, communicating elements, traffic types, latency constraints, and attenuation by in-line logic such as ④ compression, ② caching and coding.

We see various patterns in the use-case. The API server software is engaged by external requests (→) to which it replies (→). If there is an assured service-level, then these requests need to be served within specific parameters. An application's ability to serve requests at a given rate depends on factors that involve both the network and the various applications working together over it.

Having mutual awareness of application needs and network resources can enable the two to act in concert. An API server's allocation of network resources cannot prioritise requests and replies above sub-requests, otherwise this would quench the server's ability to function. While the server can have boost periods, it must eventually give way to high-priority traffic from management applications (➡) or delay-tolerant traffic for analysis (➡). Even the same class of flows can have different characters: e.g., data movement (➡) can be preemptive—from ⑥ to ⑤—and demand-driven—from ⑤ to ③.

2.2 Changing needs

There are several well-understood reasons why application needs change. For example, a common diurnal pattern involves giving priority to interactive services by day—such as API servers—and batch jobs by night.

Even within a single regular cycle there can be variations because of changing external demand. In turn, this affects the number of application instances needed to service requests. At other times, such as during planned or unplanned maintenance, instances might be wound-down and consolidated.

3 NETWORK-AWARE IN-NETWORK PROGRAMS

Network-awareness is essential to Flightplan since it needs to reason about network resources when disaggregating and placing dataplane programs. This section outlines how Flightplan works to achieve objectives set by the network operator. The next section describes how it is being extended to include application-related needs, to indirectly provide network-awareness to applications.

The increased demands on networks, particularly on datacenter networks, have spurred the development of programmable switches and the languages to program them. This has enabled faster iteration and deployment of switch functions by network operators. It also led to a proliferation of different types of hardware that are programmable using a common language such as P4 [3], including different types of smart NICs.

A feature of current dataplane programming is that an entire dataplane program is mapped to a single device. This requires the device to provide all the resources for that program to execute, and to operate at a sufficiently high rate.

Flightplan relaxes this requirement by providing an end-to-end toolchain to decompose and place P4 programs onto devices in the network, as illustrated by the bottom part of Fig. 2 which will be explained further in this section. To relax this requirement we use a suitable mix of devices instead of a single all-capable device.

Network awareness is provided to Flightplan through knowledge of the network topology and different kinds of rules.[2] One set of rules is provided for the *abstract program* that is automatically generated by Flightplan, describing the data- and control-flow between *segments* of the original P4 program. Fig. 2 shows the program's segments marked A-E, and how these are eventually allocated to different devices in the network. Abstract program rules are orthogonal to this paper's contribution, and will only be briefly described for self-containment.

[2]An example topology used in Flightplan's evaluation can be obtained from https://github.com/eniac/Flightplan/blob/master/flightplanner/examples/network_tofino.json

$$\begin{array}{l}\text{Xeon2450-1}:\\ \quad \pi_{\text{Requires}} = \left\{\text{Rate} \leq 10^{10}\right\}, \qquad \pi_{\text{Provides}} = \{\text{CPU}\},\\ \quad \text{Ports} = \left\{\ 1 \mapsto \left\{\pi_{\text{Requires}} = \left\{\text{Rate} \leq 10^{10}\right\},\ \pi_{\text{Provides}} = \{\}\right\}\ \right\}\end{array}$$

Figure 4: Flightplan rule for describing device capabilities, reproduced from the Flightplan repo. This rule names a device on the network and specifies some key properties: such as its throughput bound, and a (user-defined) proposition "CPU" that indicates its type. The rule also provides a breakdown of the capabilities of this element's network ports.

A second set of rules describes the features and capabilities of devices found in the network. An example can be seen in Fig. 4, taken from the Flightplan repo.[3] This information is used to avoid allocating segments to devices that cannot execute them at the required performance.

A final set of rules describes the cost and requirements of executing specific functions on different hardware; we call these *profiling* rules.

The different kinds of rules are processed by *Flightplanner*, a core reasoning component in Flightplan. It explores different allocations of the dataplane program's logic to different hardware in the network while ensuring that constraints are not violated, and while optimising on objectives. P4 program authors need not be aware of the deployment network's specifics, such as its topology and device composition.

4 APPLICATION-AWARE FLIGHTPLAN

This section describes how Flightplan is being extended to reason about application needs. This involves computing and checking allocation plans about network resources in a way that is more cohesive with how different applications use the network.

Fig. 2 shows how this paper builds on Flightplan. Flightplan's features are preserved and the extension captures information about application needs and integrates that information into Flightplan's reasoning process. It also seeks to capture how those needs can change, possibly as a function of changed demand. Application needs may change frequently, perhaps several times in one day. By modeling such changes we can better plan for the effects that they can have on network and device utilisation, and on overall performance.

Two changes are made to Flightplan in this extension: (i) A new form of rule is created to formulate application needs. This rule is based on application characteristics that were outlined above in §2.1 and §2.2. A stylised form of this rule is shown in Fig. 5. This is the application counterpart of the device-information rule that we saw in Fig. 4. (ii) Flightplanner is extended to use this rule. This extension consists of new inferences and checks relating to the new type of rule. So far this has not involved changing any core features in Flightplanner.

In the new type of rule shown in Fig. 5, named programs (Prog1) are given specific *modes* (**M1** in Fig. 5) that indicate different behaviours between changes (§2.2). In this way, a change is reduced to

[3]More examples can be obtained from https://github.com/eniac/Flightplan/blob/master/flightplanner/examples/devices.json

$$\begin{aligned}&\text{Prog1 mode:}\mathbf{M1}: \\ &\quad \pi_{\text{Requires}} = \left\{\text{Rate} \le 1^{10}\right\}, \quad \pi_{\text{Provides}} = \{\text{UDP}(200)\}, \\ &\quad \text{Ports} = \{(\text{Xeon2450-1}, 1)\}, \quad \text{Peers} = \left\{\text{Prog2}\right\}, \\ &\quad \text{OnPath} = \{\text{MCD\_Cache}\}\end{aligned}$$

Figure 5: Rule scheme used to capture application needs. One rule is added for each mode and application The current prototype includes 17 such rules for the use-case in §2.

a change in mode. For example, the model built for our use-case (§2) involves two modes, 'day' and 'night', to capture the different application activity during each phase.

The new rule reuses some of the notation we saw in Fig. 4 to specify the π_{Requires} and π_{Provides} sets. These are adapted to specify information about each application, such as an application's bandwidth expectations and the kind of traffic it will be generating. This information can then be cross-checked as part of a system-level analysis by Flightplanner—for instance, to check whether links are being overloaded.

The rule also adds three types of information: **Ports**: the physical ports with which applications interact on the devices on which the applications are run. This information allows cross-checking traffic patterns and bandwidth usage. **Peers**: names of peer applications with which an application interacts. This allows checking that flows do end up on network elements in which peer applications are running. **OnPath**: logic that applications can expect to find between peers. This allows checking that flows pass through elements that provide some specific logic. This is used to interface application needs with in-network computing. Given that Flightplan can now reason about both, it can also check that they are combined in desired ways. In this example we are using a Memcached cache that is part of the Crosspod [13, §2] example included with Flightplan.

5 PROTOTYPE

Flightplan was extended to use additional rules that were described above. The changes took place outside the planner's core: once plans were generated, the new rules were used to check that applications' needs could be met. Whenever an application's needs could not be met by any plan, it often turned out to be a bug in the specification of needs—for example, a wrong peer was specified. This suggested a useful role Flightplan could play to check rich cross-layer specifications of system behavior. Incorporating some of these checks within the planner's core might result in increased efficiency, and will be explored in future work. As with Flightplan, the planning is not carried out at runtime but planning can cater for different runtime scenarios.

6 DISCUSSION

This section describes initial results from using the application-aware Flightplan prototype. The prototype was applied to a network that closely followed the scenario described in §2, featuring 5 switches, 15 hosts and other hardware such as smart NICs. Previous Flightplan examples did not involve smart NICs, and our use-case involved creating a new device category for Flightplanner to use. We reused application rules for the Crosspod in-network program, which is available from the Flightplan repo.

Once the network and in-network program rules were gathered, additional rules were created to capture application needs—creating instances of the rule shown in Fig. 5. 17 instances of this rule were used to model applications in §2. Each instance describes the needs of a particular application during a specific mode (§4).

The rules were created by looking at pairs of interacting applications from Fig. 3. For example, the model includes flows between API Server to external clients, between API Server and the various KV caches, and between API Server and Storage Server. For each flow we classify their traffic type—this example includes HTTP, Memcached, rsync and others—and estimating or budgeting for applications' bandwidth needs. In this example, the number of such rules scales linearly with the number of *types* of flows between applications.

In this example we used two modes, 'day' and 'night', to model diurnal changes. Out of the 17 rules, 10 were for 'day' and 7 for 'night'. One could refine this to model multi-mode application behaviour to capture finer levels of time-intervals, or model planned maintenance or scaled-back scenarios that model outages.

7 RELATED WORK

Since writing application rules (§4) is currently done manually and relies on estimation (§6), this work could complement systems like Stroboscope [14] that can declaratively gather such measurements from the network at fine granularity. Different traffic profiles could then be classified to form different modes (§4).

Cocoon [11] and Merlin [12] use rich languages to specify and reason about access and resource requirements of networks, middleboxes and applications. Cocoon uses refinement-based modelling which can also benefit this approach. In comparison to both, this paper draws expressiveness from P4-described behaviour that can invoke middlebox-like functions.

Eden [1] provides abstractions for applications to better manage their use of the network, and enforces this at end-hosts. Dawn [15] avoids application modification by relying on annotations and effects configuration changes to switches in the network. The work in this paper does not involve application modifications, and includes P4-described in-network programs in its reasoning scope as well as applications' needs and network resources.

8 CONCLUSIONS AND FUTURE WORK

Due to its position, Flightplan can integrate applications and the network by matching between the *needs* of applications and the *resources* provided by the network. This paper describes an initial exploration on this integration. There are several open questions, forming possible directions for future work. One question concerns the model accuracy in the presence of *sharing* and *load-balancing* of application instances. Another open question involves modelling more flexible change—not only between modes, but also between node and link availability, and application mix—to enable more accurate reasoning about the detailed dynamics during these changes.

ACKNOWLEDGMENTS

I thank the NAI reviewers and the shepherd Qiao Xiang for their helpful feedback. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-19-C-0106. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

REFERENCES

[1] Hitesh Ballani, Paolo Costa, Christos Gkantsidis, Matthew P. Grosvenor, Thomas Karagiannis, Lazaros Koromilas, and Greg O'Shea. 2015. Enabling End-Host Network Functions. *SIGCOMM Comput. Commun. Rev.* 45, 4 (Aug. 2015), 493–507. https://doi.org/10.1145/2829988.2787493

[2] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. 2009. Understanding Data Center Traffic Characteristics. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking* (Barcelona, Spain) *(WREN '09)*. ACM, New York, NY, USA, 65–72. https://doi.org/10.1145/1592681.1592692

[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890

[4] Facebook Inc. 2014. Introducing mcrouter: A memcached protocol router for scaling memcached deployments. https://bit.ly/2wDbLml.

[5] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 51–66.

[6] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. ACM, New York, NY, USA, 137–152. https://doi.org/10.1145/3132747.3132756

[7] Ilias Marinos, Robert N.M. Watson, and Mark Handley. 2014. Network Stack Specialization for Performance. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 175–186. https://doi.org/10.1145/2740070.2626311

[8] Netronome Inc. 2016. Agilio CX SmartNICs. https://www.netronome.com/products/agilio-cx/.

[9] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 663–679.

[10] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. *SIGCOMM Computer Communication Review* 45, 4 (Aug. 2015), 123–137. https://doi.org/10.1145/2829988.2787472

[11] Leonid Ryzhyk, Nikolaj Bjørner, Marco Canini, Jean-Baptiste Jeannin, Cole Schlesinger, Douglas B. Terry, and George Varghese. 2017. Correct by Construction Networks Using Stepwise Refinement. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 683–698. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/ryzhyk

[12] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. 2014. Merlin: A Language for Provisioning Network Resources. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies* (Sydney, Australia) *(CoNEXT '14)*. Association for Computing Machinery, New York, NY, USA, 213–226. https://doi.org/10.1145/2674005.2674989

[13] Nik Sultana, John Sonchack, Hans Giesen, Isaac Pedisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. 2021. Flightplan: Dataplane Disaggregation and Placement for P4 Programs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 571–592. https://www.usenix.org/conference/nsdi21/presentation/sultana

[14] Olivier Tilmans, Tobias Bühler, Ingmar Poese, Stefano Vissicchio, and Laurent Vanbever. 2018. Stroboscope: Declarative Network Monitoring on a Budget. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 467–482. https://www.usenix.org/conference/nsdi18/presentation/tilmans

[15] Shuhe Wang, Dong Guo, Wei Jiang, Haizhou Du, and Mingwei Xu. 2020. Dawn: Co-Programming Distributed Applications with Network Control. In *Proceedings of the Workshop on Network Application Integration/CoDesign* (Virtual Event, USA) *(NAI '20)*. Association for Computing Machinery, New York, NY, USA, 14–19. https://doi.org/10.1145/3405672.3405808