Cost, Performance & Flexibility in OpenFlow: Pick Three

Charalampos Rotsos,* Richard Mortier,† Anil Madhavapeddy,* Balraj Singh,* Andrew W. Moore*
* University of Cambridge † University of Nottingham

***Abstract*—OS virtualization and cloud computing have radically changed the way Internet services are deployed: enterprises share third-party datacenters, deploying existing applications with minimal changes. Recent measurements reveal a lack of traffic isolation capabilities within the datacenter with network performance exhibiting high variability. We advocate addressing this problem by allowing applications to express their own forwarding logic using OpenFlow to achieve *application specific* optimal performance. We present an OpenFlow implementation within the Mirage application synthesis framework, in the form of library implementations of a modular controller and an extensible OpenFlow-enabled switch, able to expose the underlying network infrastructure to cloud applications. By linking into the application, this provides a safe yet highly extensible framework for programming network control that, although unoptimised, still provides reasonable performance when compared with existing controllers.**

I. Introduction

Recent technical advances have moved a large proportion of locally hosted enterprise services from private, in-house machine rooms to shared datacenters maintained by third party providers. This shift was enabled by the introduction of cloud computing and infrastructure virtualization. Hosting service providers now have the capability to decouple system service models from the underlying physical machine infrastructure. This allows customers to multiplex datacenter infrastructure, reducing their capital IT costs. Furthermore, cloud technologies permit fine grained control of resource allocation and provide execution isolation while placing minimal restrictions on the development environment. Finally, abstracting the service from the underlying resource allows developers to scale applications dynamically via on-demand provision of resources to running applications.

Current OS virtualization platforms such as Xen and Azure allow very fine grained control of per-application resource allocations and provide strict guarantees over CPU, memory and latency. Unfortunately however, network aspects of virtualized infrastructures remain opaque and unpredictable to application developers, a problem that manifests in two ways.

First, as recent measurement studies of cloud infrastructures show, datacenter networks provide highly variable network performance characteristics even though network resources typically remain underutilised. For example, network traces from an Azure-based datacenter show that frequent network wide congestion events occur which severely impact task execution, even though 40% of links appear to be idle even during congestion incidents [1]. Furthermore, network measurements within popular cloud computing services show that network throughput and latency exhibit high variability [2].

Second, the expressivity of current network APIs such as Berkeley Sockets is limited. As a concrete example, consider the case of a transcoding server deployed in the cloud. We can separate network traffic for such an application into two categories: flows fetching the original video and flows serving transcoded video to clients. In this example, *fetching* flows require minimum latency, to fetch the original video on time; while *serving* flows require high bandwidth, to simultaneously transmit transcoded video to multiple clients.[1]

The heart of the problem is the restricted ability to transfer information across the divide that exists between the network functionality required by datacenters and the capabilities of commodity network devices. Network switches are designed to function autonomously, with minimum administrative configuration and intervention: dynamic management of traffic is handled through relatively static link weight configuration and similar mechanisms. Although datacenter networks are under a single domain of control and could thus acquire a global view of traffic matrices to globally optimise the forwarding process, no standard mechanism exists to effectively and accurately communicate this information to running applications.

Fortunately, this mismatch between expressivity for applications and the capabilities of switches is partially addressed through OpenFlow,[2] a recently introduced software-defined network control protocol becoming widespread. OpenFlow separates the forwarding and control planes of switching devices, moving implementation of the control plane to highly programmable commodity computing platforms. The OpenFlow control plane exposes a set of network control primitives via a simple protocol over which one can implement custom forwarding logic appropriate to the demand and environment.

OpenFlow controllers are typically implemented as distinct network control applications, with one controller running in a given network. Although this has proved useful in contexts such as single-owner enterprise, we believe it can be taken further. Specifically, in multi-tenant datacenters,[3] enabling direct access to OpenFlow functionality by the application can address the underlying problem of mismatch between application control and network capability.

We present library implementations of an extensible Open-

[1] Clients may be spread across multiple network providers, so IP multicast cannot be straightforwardly applied.

[2] http://openflow.org

[3] Or even datacenters from which a single tenant provides multiple applications and services.

Flow switch implementation, and an event-driven OpenFlow controller library built on the experimental Mirage platform.[4] Using these libraries, Mirage application developers can develop cloud applications able to customise and control the flow forwarding process in real-time. We use Mirage simply as a convenient framework within which to experiment: our approach would be equally applicable using other implementation frameworks, e.g., as an OpenFlow support library with suitable interfaces hosted on Xen via *minios*.

We begin by briefly describing the current network control mechanisms in datacenters along with related work in providing guarantees and programmability in the network (§II). We then outline Mirage and describe the architecture of the OpenFlow libraries within Mirage (§III). Finally, we present a preliminary performance evaluation of our controller and switch implementations (§IV), and conclude by discussing open issues that must be addressed before our approach can be directly applied in datacenters (§V).

II. Datacenter Network Virtualization

A primary goal of datacenter networks is to minimise deployment cost. Thus, such networks are built using off-the-shelf gear, highly effective in terms of cost/performance, with upgrades usually localised and backward compatibility with existing infrastructure retained. Commodity switches sustain low prices by using general purpose, mass production silicon that provides only minimal extensibility. As a result the basic design of network switches has remained rather static through the years, generally receiving minimal incremental updates. The primary design goal for network switches remains provision of collision-free layer-2 connectivity with minimal broadcast traffic. Routers provide more hooks for control and a richer set of software protocols and interfaces, but remain expensive and relatively scarce in the datacenter.

The main mechanism currently used to provide network virtualization in datacenters is VLAN tagging [3]. This gives very good traffic isolation properties but cannot control allocation of resources such as bandwidth and latency. Furthermore, the tag field size is small and unable to scale as required in large datacenters. A more complete layer-2 virtualization solution is MPLS [4] which uses hop-by-hop labels to forward packets, strongly resembling traditional circuit switched networks with their fine-grained resource scheduling capabilities. The main drawback of MPLS is its requirement for manual configuration of paths in switches, a poor fit to the dynamic modern datacenter which must support not only largely autonomous operation, but also highly dynamic traffic patterns driven by features such as virtual machine migration.

In recent years, motivated in large part by developments in cloud computing, several authors have tried to address shortcomings in datacenter networking. For example, MPLS has been used to develop a mechanism to establish and dynamically manage paths in datacenter networks [5]. Alternatively, CamCube [6] tries to address the problem of network control by removing all network devices, providing network-wide connectivity through direct server interconnection using multi-port NICs. By implementing routing in servers, such designs provide fine level network programmability and enable development and deployment of clean slate network protocols.

Another approach that also pushes functionality to the network's edge is Multipath TCP [7]. This takes advantage of widely deployed Equal Cost Multi-Path technology to load-balance traffic over redundant links. The authors describe a deployment scenario using multipath TCP in datacenters, and present a series of experiments where the system is able to achieve higher overall throughput. Additionally, the IETF ALTO workgroup[5] is focused on the definition of a new network service, to enable application and ISP collaboration and allow optimal peer choice in P2P and CDN networks. The workgroup aim is to provide richer information to applications regarding the state and topology of the ISP network. Finally, others have proposed use of OpenFlow in the datacenter context. For example, Hedera [8] uses the OpenFlow protocol to monitor the link load of the network and dynamically reassign flows to links, using a centralised controller that spreads flows dynamically over redundant links to reduce the average load.

III. Mirage OpenFlow Implementation

We now present the details of our framework in two parts: an outline of the design goals and capabilities of the Mirage application synthesis framework (§III-A), followed by more details concerning design of the OpenFlow extension (§III-B).

A. Mirage

A key property that enabled widespread adoption of virtualization in shared infrastructures was the backward compatibility provided to existing applications. In-house maintained services could be easily migrated to the virtualized environment with a few simple configuration changes. System administrators needed only replicate data and configuration of their servers to their virtualized counterparts to embrace the revolution. Unfortunately however, as with most incrementally-deployed backward-compatible technologies, the resultant layering contains numerous redundant software components that replicate existing functionality.

For example, consider the software stack of a single application in a virtualized environment. During execution it will likely have several execution layers: the hypervisor, the guest OS, the process execution environment (typically POSIX but often also a Java VM or other relatively heavyweight language runtime), and the thread execution environment. These all provide overlapping functionality in different ways to ensure execution isolation for application code. This redundancy in functionality has two important implications.

a) Security: An important concern for the cloud applications and a commonly cited reason against wider cloud deployment: as services and software migrate to the cloud, they become vulnerable to attack by all-comers. At the same time, the increase in layers due to the cloud's architecture increases system complexity, making security evaluation harder.

[4] http://openmirage.org

[5] https://datatracker.ietf.org/wg/alto/charter/

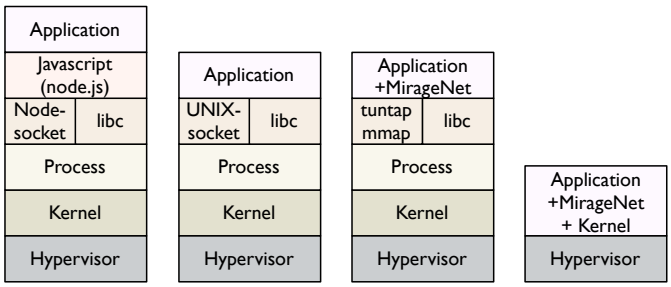

Fig. 1. Depiction of an application built to target different Mirage backends.

A growing number of current system security exploits are caused by layer complexity[6] and thus this poses a direct threat to further deployment of cloud infrastructure.

b) Efficiency: Support for legacy functionality in the OS generally wastes energy and CPU, resulting in increased cost without benefit for the hosted services. Additionally, complexity in layering means that latency due to buffering and processing may be accidentally introduced, as has already been observed to cause problems in other arenas [9].

In Mirage we address both problems through *layer collapse* via *compile-time specialisation*. For most services migrated to the cloud, the functionality required by lower layers is relatively simple (e.g., network support, filesystem access API) and can be modelled as a single generic layer of abstraction, far more simply than standard APIs such as POSIX. On the other hand, a service can be optimised during *compilation*, to generate binary-code that takes advantage of the specific attributes of its execution environment.

Mirage [10] implements these ideas, depicted in Figure 1, by enabling transparent generation of binaries for backends ranging from standalone microkernels that run directly on the Xen hypervisor, to UNIX binaries, and even JavaScript code suitable for node.js[7] or a web browser, from a single high-level source code. Applications built using Mirage become application-specific operating systems, managing their own resources either explicitly, as when running on the Xen hypervisor, or implicitly as when targeted to a UNIX process. The ability to rebuild the same code for multiple backend systems allows easy development and testing of applications: a developer builds, runs and tests his code using his standard local development environment, and then deploys to the cloud by simply compiling to the appropriate binary.

B. OpenFlow

As noted in §I, datacenter networks are currently under-utilised while still providing highly variable and often low performance to flows. The core problem can be traced to the lack of a bi-directional signalling channel between the application layer and the intermediary devices in the network.

[6]E.g., http://technet.microsoft.com/en-us/security/bulletin/ms11-083

[7]http://nodejs.org

```
1  type mac_switch = {
2    addr: OP.eaddr;
3    switch: OP.datapath_id;
4  }
5
6  type switch_state = {
7    mutable mac_cache:
8          (mac_switch, OP.Port.t) Hashtbl.t;
9    mutable dpid: OP.datapath_id list
10 }
11
12 let switch_data = {
13   mac_cache = Hashtbl.create 7;
14   dpid = [];
15 }
16
17 let join_cb controller dpid evt =
18   let dp = match evt with
19       | OE.Datapath_join c -> c
20       | _ -> invalid_arg "bogus datapath_join"
21   in
22   switch_data.dpid <- switch_data.dpid @ [dp]
23
24 let packet_in_cb controller dpid evt =
25   (* algorithm details omitted for space *)
26
27 let init ctrl =
28   OC.register_cb ctrl OE.DATAPATH_JOIN join_cb;
29   OC.register_cb ctrl OE.PACKET_IN packet_in_cb
30
31 let main () =
32   Net.Manager.create (fun mgr interface id ->
33     let port = 6633 in
34     OC.listen mgr (None, port) init
35   )
```

Fig. 2. Sample Mirage learning switch implementation.

Such a channel could convey fine-grained information regarding path load to applications, and the per-flow requirements of applications to the network devices in return. To bridge this gap we advocate direct integration of a controller into the application. Research in the field of clean-slate design has already introduced a number of systems that take advantage of OpenFlow, optimising network performance while keeping hosts unaware of the changes. Integration of the controller in the application can be considered an extension to the specialisation process at the core of Mirage.

As part of Mirage we have developed a basic OpenFlow controller library and an extensible OpenFlow-enabled switch, supporting version 1.0 of the protocol. The controller provides an event driven API inspired by NOX [11]. A sample Mirage application using the API to implement a simple learning Ethernet switch is shown in Figure 2.

In short, the main function (*l.31*) initialises a listening OpenFlow controller on the standard port 6633 (*l.33*) which calls the *init* function to initialise the switch details. This registers two event handlers: one for datapath join events (*join_cb*) and one for packet-in events (*packet_in_cb*). Each event handler is called with an event structure specific to the handled event. The current implementation supports all elementary OpenFlow messages sent from the switch to the controller. We are aware that this provides only a very low-level abstraction and we are working on developing suitable higher level events representing, e.g., link discovery or routing

protocol specific events, as provided by more mature controllers such as NOX [11].

We also provide an OpenFlow switch implementation able to operate as a guest domain bridging traffic either between virtual machines or to outgoing links. The main strength of the switch implementation is that it provides code hooks for applications to control and modify default switching functionality. This allows them to easily extend the basic OpenFlow protocol, e.g., by introducing *application specific* enhancements as OpenFlow vendor-specific messages, or extending the default flow matching tuple to include additional fields.

IV. Performance

A. Controller

We first benchmark our controller library's performance through a simple baseline comparison against two existing OpenFlow controllers, NOX and Maestro. NOX [11] is one of the first and most mature publicly available OpenFlow controllers; in its original form it provides programmability through a set of Python modules. In our evaluation we compare against both the master branch and the *destiny-fast* branch, a highly optimised version that sacrifices Python integration for better performance. Maestro [12] is an optimised Java-based controller that aims to achieve fairness among switches. We compare these against the Mirage controller targeting two different network backends: *mirage-unix* targets the UNIX Sockets backend and so uses the existing Linux TCP/IP stack, while *mirage-xen* targets the Xen hypervisor and runs as a domU virtual machine using the Mirage TCP/IP stack.

Our benchmark setup uses the *cbench* application from the Oflops benchmarking platform.[8] Each emulated switch simultaneously generates *packet-in* messages and the program measures the throughput of the controller in processing these requests. It provides two modes of operation, both measured in terms of *packet-in* requests processed per second: *latency*, where only a single *packet-in* message is allowed in flight from each switch; and *throughput*, where each switch maintains a full 64 kB buffer of outgoing packet-in messages. The first measures the throughput of the controller when serving connected switches fairly, while the second measures absolute throughput when servicing requests.

We emulate 16 switches concurrently connected to the controller, each serving 100 distinct MAC addresses. We run our experiments on a 16-core AMD server running Debian Wheezy with 40 GB of RAM and each controller configured to use a single thread of execution. We restrict our analysis to the single-threaded case as Mirage does not yet support multi-threading. For each controller we run the experiment for 120 seconds and measure the per-second rate of successful interactions. Table I reports the average and standard deviation of requests serviced per second.

Unsurprisingly, due to mature, highly optimised code, *NOX fast* shows the highest performance for both experiments. However, we note that the controller exhibits extreme short-term unfairness in the throughput test. *NOX* provides greater fairness in the throughput test, at the cost of significantly reduced performance. Maestro performs as well as NOX for throughput but significantly worse for latency, probably due to the overheads of the Java VM. Finally, Mirage throughput is somewhat reduced from NOX fast but substantially better than both NOX and Maestro with both backends; the Xen backend wins out over the UNIX backend due to reduction of layers in the network stack. In addition, Mirage Xen achieves the best product of performance and fairness among all tested controllers in the throughput test. Comparing latency, both Mirage backends perform much better than Maestro but suffer somewhat in comparison to NOX: we believe this is due to the lack of optimisation in the Mirage TCP/IP stack.

Controller	Throughput (kreq/sec)		Latency (kreq/sec)	
	avg	std. dev.	avg	std. dev.
NOX fast	122.6	44.8	27.4	1.4
NOX	13.6	1.2	26.9	5.6
Maestro	13.9	2.8	9.8	2.4
Mirage UNIX	68.1	11.7	21.1	0.2
Mirage Xen	86.5	4.4	20.5	0.0

TABLE I
OpenFlow controller performance.

B. Switch

We also use the Oflops benchmark platform [13] to evaluate performance of the Mirage switch implementation. We compare against the Open vSwitch[9] (OVS) kernel implementation, an OpenFlow-enabled software switch implemented as a Linux kernel module. OVS is currently used by many datacenter service providers to enable virtual machines to be bridged in dom0, while its OpenFlow functionality is used by vendors to implement OpenFlow firmware.

For this experiment we use two virtual machines, one running the Oflops code, the other running the OpenFlow switch configured with three interfaces bridged separately in dom0. One interface provides a control channel for the switch, while the other two are used as the switch's data channels. This represents a setup that might be used to enable an application to modify switch functionality without affecting the network functionality in dom0. Using Oflops, we generate packets on one of the data channels and receive traffic on the other, having inserted appropriate flow table entries at the beginning of the test. We run the test for 30 seconds using small packets (100 bytes) and varying the data rate.

Figure 3 plots as error boxes the min, median and max of the median processing latency of ten test runs of the experiment. We can see that the Mirage switch's forwarding performance is very close to that of Open vSwitch, even mirroring the high per-packet processing latency with a probe rate of 1 Mb/s; we believe this is due to a performance artefact of the underlying dom0 network stack. We omit packet loss due to space constraints, but can report that both implementations suffer similar levels of packet loss. However, the Mirage switch has a memory footprint of just 32 MB compared with the Open vSwitch virtual machine requirement of at least 128 MB. We are currently working toward better integration of the Mirage

[8] http://www.openflow.org/wk/index.php/Oflops

[9] http://openvswitch.org

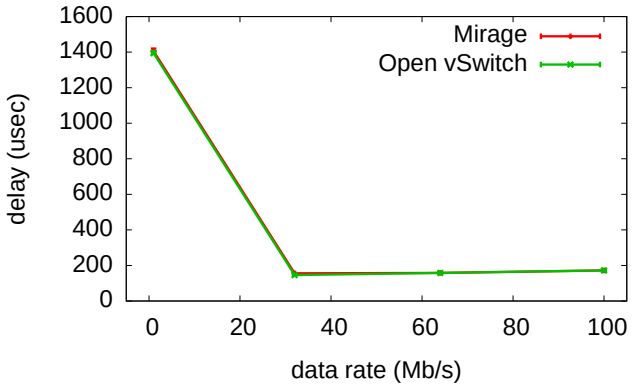

Fig. 3. Min/max/median delay when switching small (100 byte) packets using the Mirage switch and the Open vSwitch kernel module when both run as domU virtual machines.

switch functionality with the Xen network stack to achieve lower switching latency.

V. Discussion & Future Work

In this paper we have discussed the problem of network virtualization in current cloud-provider infrastructures. Motivated by existing measurements of datacenter traffic and a number of problems stemming from the current monolithic design of the underlying network, we implemented the OpenFlow protocol within the Mirage application synthesis framework to provide a network programming API integrating network control with application logic. This allows applications to exercise flow level control of their traffic through the OpenFlow protocol and so better understand and respond to the current condition of the underlying infrastructure to achieve higher, but more importantly *specifically appropriate*, network performance.

One critical key issue must be addressed before the proposed API can become a truly viable solution for datacenter networking: the direct exposure of network control to applications effectively distributes control among all entities sharing the network infrastructure. This is in direct conflict with the standard datacenter model where the service provider is firmly in charge of the control hierarchy. To address this we must extend our platform to provide a series of control mechanisms that enforce both fair use of the network by applications and symbiotic coexistence of application forwarding logic with the control policies of the service provider. We are currently examining how such features can be implemented through OpenFlow-aware software filters, and can thus potentially offer a richer economical model able to increase the cost for applications that want to utilize more network resources such as flow table entries.

Another area of ongoing work is in the abstraction provided by our API. In its current form it directly replicates all the semantically important OpenFlow messages, which provides a rather low level API. We believe most applications could benefit from transformation of the information provided by the OpenFlow protocol into higher level semantics. Frameworks like NetCore [14] or SFNet [15] describe novel control frameworks that use the OpenFlow protocol directly, but provide higher level functionality. For example, NetCore enables administrators to deploy policies while the NetCore language ensures that the policy is effective at any given point in time. Similarly, more advanced language integration provided in systems such as Nettle [16] and Frenetic [17] brings significant benefits in terms of the ease with which controller code can be composed and reasoned about.

To conclude, the current socket API remains a basically useful abstraction for cloud computing application development, as it is widely used and understood. However, we believe that future datacenter network APIs should provide extended semantics to developers so that they may characterise the expected performance of flows in more detail. This information could then be translated in real-time by the underlying OpenFlow library into the required OpenFlow commands.

References

[1] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *Proc. 9th ACM Internet Measurement Conference (IMC)*. New York, NY, USA: ACM, 2009, pp. 202–208.

[2] A. Li, X. Yang, S. Kandula, and M. Zhang, "CloudCmp: comparing public cloud providers," in *Proc. 10th ACM Internet Measurement Conference (IMC)*. Melbourne, Australia: ACM, 2010, pp. 1–14.

[3] *Virtual Bridged Local Area Networks*, IEEE, May 2006, IEEE Standard 802.1Q, 2005 edition.

[4] E. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol Label Switching Architecture," IETF, RFC 3031, Jan. 2001.

[5] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang, "SecondNet: a data center network virtualization architecture with bandwidth guarantees," in *Proc. 6th International Conference on Emerging Networking Experiments and Technologies (CoNext)*. Philadelphia, PA: ACM, 2010, pp. 1–12.

[6] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O'Shea, and A. Donnelly, "Symbiotic routing in future data centers," in *Proc. ACM SIGCOMM 2010*. New York, NY, USA: ACM, 2010, pp. 51–62.

[7] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath TCP," *SIGCOMM CCR*, vol. 41, pp. 266–277, Aug. 2011.

[8] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: dynamic flow scheduling for data center networks," in *Proc. 7th USENIX Conference on Networked Systems Design and Implementation (NSID)*. San Jose, California: USENIX Association, 2010.

[9] J. Gettys, "Bufferbloat: Dark buffers in the Internet," *IEEE Internet Computing*, vol. 15, no. 3, p. 96, May 2011.

[10] A. Madhavapeddy, R. Mortier, R. Sohan, T. Gazagnaire, S. Hand, T. Deegan, D. McAuley, and J. Crowcroft, "Turning down the lamp: software specialisation for the cloud," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing (HotCloud)*. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11.

[11] 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.*, vol. 38, pp. 105–110, July 2008.

[12] Z. Cai, A. L. Cox, and T. S. E. Ng, "Maestro: A system for scalable OpenFlow control," Rice University, Tech. Rep. TR-10-11.

[13] C. Rotsos, N. Sharrar, S. Uhlig, R. Sherwood, and A. W. Moore, "OFLOPS: An open framework for OpenFlow switch evaluation."

[14] C. Monsanto, N. Foster, R. Harrison, and W. David, "A compiler and run-time system for network programming languages," ser. Symposium on Principles of Programming Languages (POPL), 2012.

[15] K.-K. Yap, T.-Y. Huang, B. Dodson, M. S. Lam, and N. McKeown, "Towards software-friendly networks," in *Proceedings of the first ACM Asia-Pacific Workshop on Systems (APSys)*. New York, NY, USA: ACM, 2010, pp. 49–54.

[16] A. Voellmy and P. Hudak, "Nettle: Taking the sting out of programming network routers," in *Proceedings of 13th International Symposium on Practical Aspects of Declarative Languages*, ser. Lecture Notes in Computer Science, vol. 6539. Austin, TX, USA: Springer, Jan. 24–25 2011, pp. 235–249.

[17] N. Foster, R. Harrison, M. J. Freedman, C. Monsonto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *Proc. ACM ICFP '11*, Tokyo, Japan, Dec. 2011.