Corybantic: Towards the Modular Composition of SDN Control Programs

Jeffrey C. Mogul*§, Alvin AuYoung†, Sujata Banerjee†, Lucian Popa†,
Jeongkeun Lee†, Jayaram Mudigonda*‡, Puneet Sharma†, Yoshio Turner†
†HP Labs, Palo Alto, *Google, Inc.
†{FirstName.LastName}@hp.com, §jeffmogul@acm.org, ‡jmudigonda@google.com

ABSTRACT

Software-Defined Networking (SDN) promises to enable vigorous innovation, through separation of the control plane from the data plane, and to enable novel forms of network management, through a controller that uses a global view to make globally-valid decisions. The design of SDN controllers creates novel challenges; much previous work has focused on making them scalable, reliable, and efficient.

However, prior work has ignored the problem that multiple controller functions may be competing for resources (e.g., link bandwidth or switch table slots). Our *Corybantic* design supports modular composition of independent controller modules, which manage different aspects of the network while competing for resources. Each module tries to optimize one or more objective functions; we address the challenge of how to coordinate between these modules to maximize the overall value delivered by the controllers' decisions, while still achieving modularity.

Categories and Subject Descriptors

C.2.3 [**Network Operations**]: Network Management

General Terms

Design, Economics, Management

Keywords

Software-Defined Networking

1. INTRODUCTION

Software Defined Networking (SDN) promises a better approach, not just for network innovation, but also for managing a network. By separating the control plane from the data plane, via a protocol such as OpenFlow, SDN allows the use of an arbitrarily complex, *central* controller that can dynamically make decisions that can maintain global policies and objectives.

Of course, the promise of SDN relies on the creation of reliable, scalable, and efficient controller software. Many researchers and developers have worked on this problem (e.g., [5, 14, 21, 20]).

For a production network, however, an SDN controller must simultaneously respect many concerns, possibly with competing objectives. For example, a power manager's objective might be to re-route traffic onto a smaller set of links, so that it can turn off a switch, while a QoS manager might want to maintain as many live links as possible, to ensure that its SLAs are met. Other examples of objectives include routing, flow-level traffic engineering, support for fast failover, middlebox insertion, VM migration, etc.

One might build a monolithic controller that can juggle all of these issues, but we believe that without modularity, such a controller will be hard to build, maintain, and extend, and perhaps just as resistant to further innovation as traditional hardware-defined networks.

In this paper, we propose a controller framework design, *Corybantic*, that focuses on both achieving modular composition and maximizing the overall value delivered by the controller's decisions. Our goal is for modules to expose sufficient information about their local objectives and policies so that, operating collaboratively through Corybantic, they maximize system-wide objectives while meeting all of their policy constraints. At the same time, the inter-module interfaces should be as narrow as possible, to support modularity and composition.

We contrast our work with Frenetic [8, 9] and Pyretic [18]: both systems support composition of modules by resolving conflicts over specific OpenFlow rules. Corybantic supports composition of modules that might conflict over higher-level policies and objectives. In other words, Frenetic, Pyretic, and all the prior work that we know of, aims at better ways to get the network to *do* what you want it to do; Corybantic, in contrast, is aimed at *deciding* what you want it to do. As a consequence, Corybantic operates at a higher layer of the controller stack, and is mostly complementary to prior work.

Hotnets '13, November 21–22, 2013, College Park, MD, USA.

Solving this kind of constrained multi-objective optimization problem can be difficult. Corybantic sidesteps several hard problems. First, it converts a multi-objective problem into a single-objective problem by expressing objectives in a common currency. Second, it explores the use of fast heuristics, with an iterative, domain-specific approach that explicitly represents topology to improve allocation decisions quickly, instead of requiring an optimal solution.

Corybantic is a work in progress. We have an early implementation of various modules and present an initial evaluation of the composition of these modules.

2. CONTEXT

Corybantic could be useful, for example, in a cloud (Infrastructure-as-a-Service) network, which provides service to an ever-changing set of users, each of whom is explicitly charged for the services they request, and whose needs vary dynamically. Any such network must consider its costs and revenues when making policies governing resource allocation. For simplicity, we assume that a provider's revenues come as explicit payment from its users[1].

A network provider would start by choosing a set of modules (e.g., from a "module store") as appropriate for its specific network. We would not expect any network to use all of the possible modules. We assume that there is just one network provider, so the controller modules are not malicious.

2.1 Per module policies and objectives

Our vision is that the central controller can integrate the opportunities provided by its constituent modules to approximate a globally-optimal objective, while meeting all of the policies (e.g., firewall rules) imposed by these modules.

Each individual module will have its own policies and objectives. Policies express constraints on what is allowed; objectives express the costs and benefits of specific controller actions. Consider, for example, a QoS-control module. Its objectives include maximizing the revenue from bandwidth promises made to customers; one of its policies might be "never renege on a promise."

We believe that, in order to compare costs and benefits across a range of independent modules, the controller as a whole must express these using a common currency. AuYoung *et al.* [2] point out the value of using money as a single metric for expressing "all the QoS-'ilities' that matter in a particular circumstance." The actual currency units do not matter; AuYoung *et al.* chose to use "florins."

2.2 Examples of controller modules

To illustrate our motivation for supporting composition of controllers with competing policies and objectives, we briefly sketch some examples, focused on data-center networks. All of these examples have been described previously as independent controllers:

- **Bandwidth allocator**: A module that allocates guaranteed bandwidth to a set of endpoints. Examples include Gatekeeper [19], Oktopus [3], and many others.
- **Flow latency controller**: A module to support end-to-end latency bounds for flows or flow classes (as by Kim *et al.* [13].
- **Flow-level traffic engineering**: A module that re-routes flows to achieve an objective, like load balance (as in Hedera [1]).
- **VM-migrator**: A module that migrates VMs between servers, e.g., for consolidation (as in SecondNet [10], Virtue [16], etc).
- **Power control manager**: A module that reduces energy costs by attempting to turn off under-utilized resources (as in ElasticTree [12]).
- **Bypass-link controller**: A module, such as Flyways [11], that looks for over-utilized paths and opportunities to create wireless bypasses. Similarly, Helios [6] looks for opportunities to create optically-switched high-bandwidth bypasses.

3. DESIGN OF CORYBANTIC

A Corybantic system consists of a set of modules, which might vary among different installations, and a central coordinator function, all running on top of a typical SDN controller, as shown in Fig. 1(a). Our goal is to allow multiple controller modules to operate in concert, rather than pulling in opposing directions. The basic insight behind Corybantic is that the use of a common currency can allow modules to express high-level objectives independently, and a central coordinator to resolve potential conflicts.

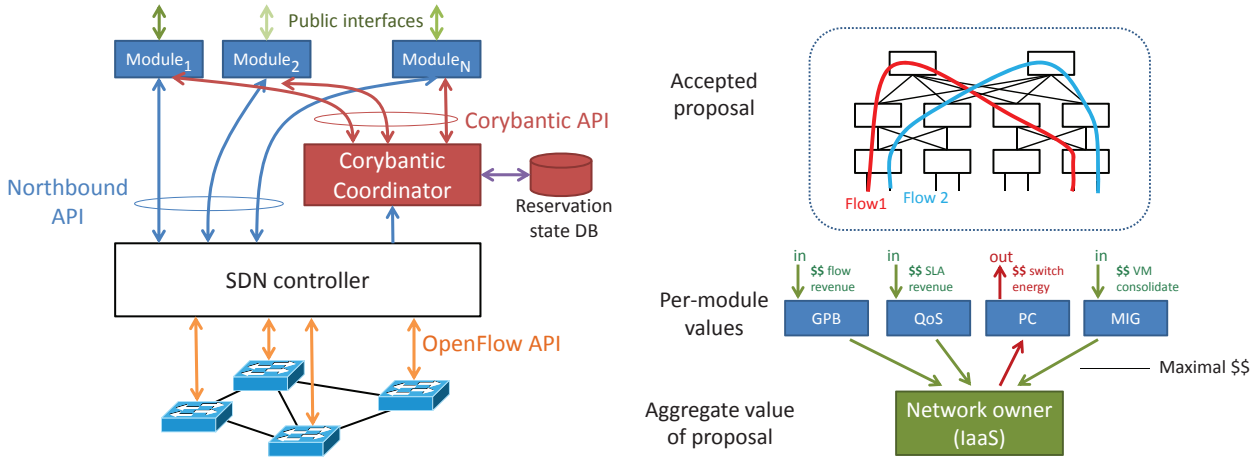
3.1 Expressing objectives

A network has an underlying physical topology. Corybantic represents the physical topology of the network as a graph of resources including switches and links. Some of these resources can be subdivided into virtual slices; e.g., a 10Gbit link could be sliced, using rate limiters, into a set of slower virtual links. Moreover, each of these resources has a reservation state representing what (share) belongs to each tenant.

We find it natural to express module goals in terms of virtual subset topologies (possibly a disconnected graph) of the underlying network topology, rather than in terms of only the specific collection of individual link and switch resources relevant to each module. For example, the current goals of a bandwidth allocator module could be satisfied by one of several collections of network paths formed from slices of the physical links. The goals of an energy manager could be satisfied by any of several subset topologies that exclude specific switches or links, which can then be powered off.

We can thus view Corybantic as a means to agree on a single virtual subset topology that is acceptable with respect

[1]We have not carefully considered whether our approach applies to a wider set of businesses, such as ISPs, or those where IT costs and benefits are not well-quantified.



(a) Relationship between Corybantic and SDN controller.

(b) Example proposal and value flows in Corybantic.

Figure 1: Context for Corybantic.

to the goals of all modules. (Because "virtual subset topology" is unwieldy, generally in this paper we use "topology" to imply that term.)

3.2 An iterative approach

Do we first look for a topology that meets the customer demands, or do we first find a placement of demand that assumes a specific topology? Exploring all of the possible options seems like it would result in an infeasibly-large search space. To avoid this chicken-or-egg problem, Corybantic uses a multi-phase iterative approach.

The Corybantic coordinator proceeds in periodic rounds. In each round, the coordinator and its modules execute four phases:

- **Phase 1 – some modules propose topology changes**: A proposal is a change in the reservation state or activity (on/off) of a resource in the topology. Each module may make one or more proposals to modify the topology. A "topology" change may involve turning servers, switches, or links on or off, adding a switch table entry, or moving VMs.These proposals should be "small" deltas from the current state, not huge changes.

  Making good proposals is the key challenge for Corybantic. We discuss this key challenge in Sec. 4.1.
- **Phase 2 – each module evaluates every current proposal** generated in phase 1, assigning a value to it (top of Fig. 1(b)). The value considers the benefits that the module gains from a proposal, the costs imposed on the module, including the cost of making changes (e.g., moving a VM) and any "costs" created by unfairness. For example, a QoS controller would express value as the revenue collected by the flows it can support within the proposal. Some modules might give a negative value to a proposal.

  By distributing the task of proposal evaluation to all of the modules in the system, we modularize the computation of the values that derive from various points of view. For example, the power module only needs to consider the electricity-related costs and benefits of turning off a switch, while the QoS-related benefits (which might be negative) are measured entirely within other modules.

  Also, by separating the proposal-generation phase from the proposal-evaluation phase, we relieve the proposal generators from having to understand the value models of other modules.

  Fig. 1(b) illustrates how values flow in Corybantic. For concreteness, we assume that all values are expressed in terms of dollars (or some other real-world currency), to reflect the grounding of these values in the real-world money flowing into and out of the provider.
- **Phase 3 – the coordinator picks the best proposal**: An overall coordinator function assigns a global value to each proposal, for example, a sum over the phase-2 values and picks the best proposal (if any have positive values).
- **Phase 4 – the modules instantiate the chosen proposal**: Once the coordinator has chosen a proposal, any module that is affected by that proposal is directed to instantiate it. This might mean changing a topology element, moving a VM, or perhaps just informing a user that its VMs now have more or less access to spare capacity, so that the user's application can adapt.

The goal of this iterative loop is to constantly adapt to new customer demands, not to focus on convergence to a fixed point.

3.3 Alternative approaches

Casting the search for improved value as an iteration avoids having to impose an *a priori* ordering on the modules; any fixed ordering would over-determine the pruning of the search space, and inhibits modularity. Instead, we express "priority" not as an explicit property of a module,

but through the valuations placed on its specific proposals. This allows dynamic changes in the relative merits of assigning a resource to one module or another. For example, load balance might be more important during peak hours, while power reduction might matter more during off-peak hours, as reflected entirely in the values that these modules place on proposals during those periods (rather than via static module priorities.)

4. DESIGNING MODULES

The success of Corybantic will depend on the design of the controller modules. Ideally a module generates both *high-value* proposals to deliver value towards its objective function, as well as *a variety* of proposals to allow better coverage of the allocation space.

4.1 Making good proposals

What makes a proposal "good"? We list some high-level principles:

Make small proposals, but not tiny ones: If a proposed topology change is too large, most likely the other modules in the system will give it a very negative value, either because it disrupts their own needs too much, or the cost of change is too high. Modules should propose changes that are small enough to be acceptable. However, there is little point in making a proposal that is too small: it should not leave the network mid-way between useful states. For example, if a VM-migrator module wants to move 10 VMs away from a server, a proposal that moves just one VM probably has little or no value.

Make a reasonable number of proposals: Providing multiple topology options gives the coordinator more options to evaluate and should lead to faster convergence to a useful state.

Offer variety: When making multiple proposals, don't make them too similar; otherwise, the benefits of having more options to evaluate would not justify the extra cost of evaluating them.

Don't give up too late or too soon: A module should build upon iterations to make informed counter proposals that differ enough from rejected proposals and relate to higher-valued proposals.

4.2 Search strategies

During each round, a control module may need to continually generate a counter-proposal, or look for other opportunities to meet its objectives. This search mechanism, if naively implemented, risks locking onto local optima. There are two basic approaches to avoid this problem.

The first approach is inspired by search heuristics often used in genetic algorithms: occasional jumps can be injected into the search space to avoid local optima. For example, the coordinator can inject occasional jumps in the current allocation. The coordinator decides when an iteration round should make a jump, of what magnitude, and (prior to phase 1) informs modules that they should offer "big-delta" proposals with some probability. Likewise, each module can inject such jumps within its own proposal generation search in order to provide such big-delta proposals.

The second approach avoids local optima by carefully crafting a module's objective function. If it can define an objective function that is convex, it guarantees convergence in its search heuristic. Indeed, well-known methods, such as gradient descent, are often used in implementations of solving these optimization functions [4]. Using a convex function may require a trade-off between optimality (i.e., since the convex function may merely be a proxy for the original optimization function) and convergence.

These two techniques are complementary. Balancing the value delivered by proposals and the risks of search oscillations or local optima may require further investigation.

There is no magic formula for implementing a module that obeys these principles. We realize that it might be quite difficult, in practice, to engineer a good proposal generator, and that this may require domain-specific engineering – each module may be quite different.

5. MODULE PROTOTYPES

Thus far, we have implemented prototypes of several Corybantic controller modules. These modules are written in Python in order to eventually interface with the suite of available OpenFlow software that are also implemented in Python [15, 18]. It is important to note that these prototypes represent proof-of-concepts based on control modules in the literature, but likely require more detail if deployed in a real network.

5.1 Guaranteed Pipe-Bandwidth Module

The Guaranteed Pipe-Bandwidth (GPB) module provides minimum bandwidth guarantees for tenants in a shared data center. Tenants are customers who pay for VM hosting on the data center, and whose VMs require a minimum intra-network bandwidth guarantee. Similar modules might support "hose-model" guarantees, which cover the aggregate bandwidth into and out of one endpoint (e.g., Gatekeeper [19]), or other guarantee forms (e.g., Virtual Oversubscribed Clusters [3]), but a pipe-model module is the easiest to describe and implement.

Evaluation function: GPB values a proposal based on the expected income it will receive for guarantees it has given to customers. This expected income may include the net present value of expected *future* customer requests, as well as the potential penalty (i.e., negative value) for being unable to honor a previously guaranteed request, or for having to update routes and rate limiters to shift traffic.

Making Proposals: Currently, GPB attempts to get all resources and satisfy all flows (active flows, pending flows and projected flows) in the flow table.

5.2 Bandwidth Availability Module

The Bandwidth Availability Module (BAM) tries to conserve core bandwidth by re-routing traffic[2]. This module is motivated by bandwidth-constrained environments and is based upon the requirements described by Bodik *et al.* [4].

Evaluation function: BAM considers the value of bandwidth it conserves. It measures (and in some cases, estimates), the client traffic that traverses the network core and tries to minimize this value. Based on the same information used by GPB to place a dollar value on bandwidth, the amount of available bandwidth "slack" in the core is the metric used to evaluate a proposed topology.

Making proposals: BAM attempts to re-route traffic by migrating VMs that generate and receive traffic. This module uses a procedure similar to that described by Bodik *et al.* to cluster the machines in racks within clusters based upon a minimum k-cut, where k is the number of available edge switches in the network.

5.3 Fault Tolerance Module

The Fault Tolerance Module (FTM) tries to increase the worst-case survivability (WCS) of a tenant's hosted services. We borrow the definition and model of WCS directly from Bodik *et al.* as the fraction of hosted services (i.e., VMs) that survive any single failure in the network.

Evaluation function: FTM calculates the average WCS of an allocation across all services. Based on the WCS, it computes the revenue lost from the host services due to expected downtime. We expect the values required for these calculations to be provided externally to the module. Moreover, we expect that these measurements are typical bookkeeping for a typical IaaS provider.

Making proposals: At a high level, FTM attempts to "spread" VMs across failure domains. Proposals are made as long as improvements can continue, for example, until all VMs are in separate fault domains. Each proposal is greedy in that it attempts to move a single VM at a time to a different fault domain, when it resides in the same fault domain as another.

6. SIMULATIONS

In this section, we evaluate the performance and convergence of our BAM and FTM controllers. As described in Section 5, BAM attempts to minimize core bandwidth consumption, and FTM attempts to increase the fault tolerance of tenant applications. As described by Bodik *et al.*, these objectives are directly at odds [4].

In order to balance these competing concerns, Bodik *et al.* define a single optimization function that is parameterized by the variable α. α, in essence, establishes a weighted priority between these two concerns and defines how these competing objectives should be resolved.

[2]In our implementation, it accomplishes this by moving the source and destination VMs generating traffic.

We argue that setting α correctly requires detailed understanding of how each of these concerns relates to the global value delivered to the system. Accordingly, we have implemented the BAM and FTM modules independently to relate their individual optimization functions to the revenue model in the network (Table 1). Note that this definition is merely for illustrative purposes and can easily be adapted for a more realistic scenario.

α	**Equivalent Corybantic definition**
$0 < \alpha < 1/2$	A unit loss of availability costs $\$\alpha$ more than an equivalent unit loss in core bandwidth.
$\alpha = 1/2$	Availability and core bandwidth are equally important.
$1/2 < \alpha$	A unit loss of availability costs $\$\alpha$ less than an equivalent unit loss in core bandwidth.

Table 1: Interpretation of α for FTM and BAM Corybantic modules.

Using these modules, we simulate the Corybantic allocation process in a simple network topology with two aggregation switches and four racks, with each rack containing a maximum number of 4 VMs. The simulation began with an initial random allocation and the numbers reflect an average over 10 runs.

Table 2 compares the performance of FTM and BAM to an optimal, monolithic controller for various values of α. We did a straightforward offline calculation for the specific topology and workload to obtain this optimal value.

We see that in most of these settings, the values converge within a few rounds, and the value delivered by the allocation is within 10% of optimal in most cases. These standard deviation for total value is less than 10% of the average; the standard deviation for the number of rounds until convergence is less than 1 round.

These instances correspond to larger and smaller values of α. This result is due to the fact that in such cases, the convergence is heavily dependent on the search path of a single module's objective function.

The instances where convergence does not result in the optimal value can be improved with simple search heuristics (i.e., genetic algorithm-based "jumps", as mentioned in Section 4.2). We do not know if the effectiveness of these techniques generalizes, as these are simply preliminary results.

α	**Average fraction of optimal value**	**Average rounds until convergence**
1/5	0.93	2
1/4	0.91	2.4
1/3	0.83	1
1/2	0.83	1.2
2/3	1.0	1.4
4/5	1.0	1.3

Table 2: Average quality and speed of allocation when composing FTM+BAM modules.

7. ADDITIONAL DESIGN ISSUES

In this section, we discuss additional design issues that we have considered, but have not fleshed out in implementation.

7.1 Enforcing constraints

In emergencies, "goals" might become constraints: for example, a thermal overload must be resolved immediately, even at the cost of violating some network performance guarantees. Other constraints might be tagged to indicate that they can be temporarily violated during emergencies.

7.2 Time-related issues

So far, we have ignored how Corybantic deals with the concept of time. We need to explicitly consider time for several reasons:

- **Implementation delays**: Many proposals cannot be instantiated instantaneously. For example, a VM-migrator might take several minutes to move a large VM. Thus, proposals made in phase 1 must specify the expected delay-to-instantiate.
- **Deadlines**: Conversely, some modules might tag their proposals with deadlines. For example, a VM-migrator that needs to respond to thermal overloads may require very rapid changes.
- **Evaluation time**: Because Corybantic proceeds in small steps, each round must finish quickly. Obtaining a short cycle time may require some careful optimization. For example, since proposals are expressed as deltas, the evaluates method in modules might use an incremental algorithm, rather than re-computing the value over the entire proposed topology.
- **Time-dependent valuations**: Suppose a QoS module wants to admit a high-bandwidth flow that will last for 2 minutes. If making room for this flow incurs a cost, such as turning on a switch, that will last for at least 5 minutes, the benefit of admitting the flow may not justify this cost.

Therefore, proposals and evaluations must both include time-frames over which they are desirable or valid.

7.3 Security

Corybantic does not, as of now, provide any inter-module security mechanisms. In this respect, it does not differ from a monolithic controller implementation with the same functionality. All of the modules are installed by, and run on behalf of, the provider, as is the case with most existing system and network management tools. Therefore, just as with a monolithic controller, the module implementers will attempt to cooperate with each other to achieve the best possible result, rather than try to hog resources or to game Corybantic's evaluation function.

In a system where some modules are provided by third parties, it would be useful to protect the modules from each other, and (more important) to prevent rogue or buggy modules from interfering with resource allocation. We can treat this as a motivation for good debugging support (see §7.4), or as an opportunity for future work – for example, the Coordinator might impose quotas on the resources that a third-party module could obtain.

7.4 Debugging

Networks, even with central controllers, are complex, dynamic beasts; they will have bugs. Corybantic will not eliminate bugs; any such system should have debugging support built into its design.

Corybantic's use of modularity can prevent invisible interactions between modules, but even the "visible" interactions (exposed through the proposal and evaluation phases) can lead to bugs. And, as we have already noted, designing modules to generate "good" proposals could be challenging. Because all interactions between modules are exposed to Corybantic in the proposal and evaluation phases, the system can therefore expose, in a debugging console, details such as which modules propose which topologies, the valuations assigned to each proposal, any policy violations, or why a controller is otherwise not behaving as expected (e.g., from human observation).

8. RELATED WORK

Most of the prior work on SDN controllers has focused either on enabling more flexible and rapid innovation (e.g., Frenetic [8, 9], Pyretic [18] and Trema [21]), or on improving the performance or scalability of the controller platform (e.g., Maestro [5], Mirage [20], HyperFlow [22], and Onix [14].)

Ford [7] observed that non-transparent layering of multiple control functions, between cloud-provider controllers and cloud-tenant controllers, can lead to "potentially catastrophic" interactions over shared resource dependencies. Corybantic might not directly resolve that, but perhaps tenants could benefit from a narrow interface into the provider's Corybantic system, analogous to the Mirage interface proposed by Rotsos *et al.* [20].

Network economists have explored related issues (e.g., [17]) but have focused on obtaining accurate prices, not on software modularity.

9. REFERENCES

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

[2] A. AuYoung, L. Grit, S. Wiener, and J. Wilkes. Service contracts and aggregate utility functions. In *Proc. HPDC*, pages 119–131, 2006.

[3] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *Proc. SIGCOMM*, pages 242–253, 2011.

[4] P. Bodk, I. Menache, M. Chowdhury, P. Mani, D. A. Maltz, and I. Stoica. Surviving failures in

bandwidth-constrained datacenters. In *Proc. SIGCOMM*, pages 431–442. ACM, 2012.

[5] Z. Cai, A. L. Cox, and T. S. E. Ng. Maestro: Balancing Fairness, Latency and Throughput in the OpenFlow Control Plane. Tech. Rep. TR11-07, Rice Univ., 2011.

[6] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. In *Proc. SIGCOMM*, 2010.

[7] B. Ford. Icebergs in the Clouds: the Other Risks of Cloud Computing. In *Proc. HotCloud*, 2012.

[8] N. Foster, M. J. Freedman, R. Harrison, J. Rexford, M. L. Meola, and D. Walker. Frenetic: A High-Level Language for OpenFlow Networks. In *Proc. PRESTO*, pages 6:1–6:6, 2010.

[9] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *Proc. ICFP*, pages 279–291, 2011.

[10] 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. Co-NEXT*, 2010.

[11] D. Halperin, S. Kandula, J. Padhye, P. Bahl, and D. Wetherall. Augmenting Data Center Networks with Multi-Gigabit Wireless Links. In *Proc. SIGCOMM*, pages 38–49, 2011.

[12] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. ElasticTree: Saving energy in data center networks. In *Proc. NSDI*, 2010.

[13] W. Kim, P. Sharma, J. Lee, S. Banerjee, J. Tourrilhes, S.-J. Lee, and P. Yalagandula. Automated and Scalable QoS Control for Network Convergence. In *Proc. INM/WREN*, Apr. 2010.

[14] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proc. OSDI*, 2010.

[15] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, 2010.

[16] L. Liu, H. Wang, X. Liu, X. Jin, W. B. He, Q. B. Wang, and Y. Chen. GreenCloud: A New Architecture for Green Data Center. In *Proc. ICAC-INDST*, pages 29–38, 2009.

[17] B. Lubin, A. Juda, R. Cavallo, S. Lahaie, J. Shneidman, and D. C. Parkes. ICE: An Expressive Iterative Combinatorial Exchange. *J. Artificial Intelligence Research*, 33:33–77, 2008.

[18] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software-Defined Networks. In *USENIX NSDI*, 2013.

[19] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. Guedes. Gatekeeper: supporting bandwidth guarantees for multi-tenant datacenter networks. In *Proc. WIOV*, 2011.

[20] C. Rotsos, R. Mortier, A. Madhavapeddy, B. Singh, and A. W. Moore. Cost, Performance & Flexibility in OpenFlow: Pick Three. In *Proc. IEEE SDN Workshop*, 2012.

[21] H. Shimonishi, Y. Chiba, Y. Takamiya, and K. Sugyo. Trema: An Open Source OpenFlow Controller Platform. In *GEC-11 Poster*, 2011.

[22] A. Tootoonchian and Y. Ganjali. HyperFlow: A Distributed Control Plane for OpenFlow. In *Proc. INM/WREN*, Apr. 2010.