Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications

Soheil Hassas Yeganeh
University of Toronto
soheil@cs.toronto.edu

Yashar Ganjali
University of Toronto
yganjali@cs.toronto.edu

ABSTRACT

Limiting the overhead of frequent events on the control plane is essential for realizing a scalable Software-Defined Network. One way of limiting this overhead is to process frequent events in the data plane. This requires modifying switches and comes at the cost of visibility in the control plane. Taking an alternative route, we propose Kandoo, a framework for preserving scalability without changing switches. Kandoo has two layers of controllers: (*i*) the bottom layer is a group of controllers with no interconnection, and no knowledge of the network-wide state, and (*ii*) the top layer is a logically centralized controller that maintains the network-wide state. Controllers at the bottom layer run only *local* control applications (*i.e.*, applications that can function using the state of a single switch) near datapaths. These controllers handle most of the frequent events and effectively shield the top layer. Kandoo's design enables network operators to replicate local controllers on demand and relieve the load on the top layer, which is the only potential bottleneck in terms of scalability. Our evaluations show that a network controlled by Kandoo has an order of magnitude lower control channel consumption compared to normal OpenFlow networks.

Categories and Subject Descriptors

C.2 [**Computer-communication networks**]: Network Architecture and Design

Keywords

Software-Defined Networks, Data Center Networks, Distributed Control Plane

1. INTRODUCTION

Frequent and resource-exhaustive events, such as flow arrivals and network-wide statistics collection events, stress the control plane and consequently limit the scalability of OpenFlow networks [5, 18, 4]. Although one can suppress flow arrivals by proactively pushing the network state, this approach falls short when it comes to other types of frequent events such as network-wide statistics collection. Current solutions either view this as an intrinsic limitation or try to address it by modifying switches. For instance, HyperFlow [18] can handle a few thousand events per second, and anything beyond that is considered a scalability limitation. In contrast, DIFANE [21] and DevoFlow [5] introduce new functionalities in switches to suppress frequent events and to reduce the load on the control plane.

To limit the load on the controller, frequent events should be handled in the closest vicinity of datapaths, preferably without modifying switches. Adding new primitives to switches is undesirable. It breaks the general principles of Software-Defined Networks (SDNs), requires changes to the standards, and necessitates the costly process of modifying switches. Thus, the important question is: "*How can we move control functionalities toward datapaths, without introducing new datapath mechanisms in switches?*"

To answer this question, we focus on: (*i*) environments where processing power is readily available close to switches (such as data center networks) or can be easily added (such as enterprise networks), and (*ii*) applications that are local in scope, *i.e.*, applications that process events from a single switch without using the network-wide state. We show that under these two conditions one can offload local event processing to local resources, and therefore realize a control plane that handles frequent events at scale.

We note that some applications, such as routing, require the network-wide state, and cannot be offloaded to local processing resources. However, a large class of applications are either local (*e.g.*, local policy enforcer and Link Layer Discovery Protocol [2]) or can be decomposed to modules that are local (*e.g.*, elephant flow detection module in an elephant flow rerouting application).

Kandoo. In this paper, we present the design and implementation of Kandoo, a novel distributed control plane that offloads control applications over available resources in the network with minimal developer intervention and without violating any requirements of control applications. Kandoo's control plane essentially distinguishes *local control applications* (*i.e.*, applications that process events locally) from *non-local applications* (*i.e.*, applications that require access to the network-wide state). Kandoo creates a two-level hierarchy for controllers: (*i*) *local controllers* execute local applications as close as possible to switches, and (*ii*) a logically centralized *root controller* runs non-local

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
HotSDN'12, August 13, 2012, Helsinki, Finland.
Copyright 2012 ACM 978-1-4503-1477-0/12/08 ...$15.00.

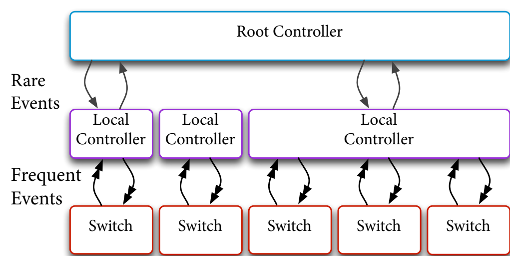


Figure 1: Kandoo's Two Levels of Controllers. Local controllers handle frequent events, while a logically centralized root controller handles rare events.

control applications. As illustrated in Figure 1, several *local controllers* are deployed throughout the network; each of these controllers controls one or a handful of switches. The root controller, on the other hand, controls all local controllers.

It is easy to realize local controllers since they are merely switch proxies for the root controller, and they do not need the network-wide state. They can even be implemented directly in OpenFlow switches. Interestingly, local controllers can linearly scale with the number of switches in a network. Thus, the control plane scales as long as we process frequent events in local applications and shield the root controller from these frequent events. Needless to say, Kandoo cannot help any control applications that require network-wide state (even though it does not hurt them, either). We believe such applications are intrinsically hard to scale, and solutions like Onix [8] and HyperFlow [18] provide the right frameworks for running such applications.

Our implementation of Kandoo is completely compliant with the OpenFlow specifications. Data and control planes are decoupled in Kandoo. Switches can operate without having a local controller; control applications function regardless of their physical location. The main advantage of Kandoo is that it gives network operators the freedom to configure the deployment model of control plane functionalities based on the characteristics of control applications.

The design and implementation of Kandoo are presented in Sections 2. Our experiments confirm that Kandoo scales an order of magnitude better than a normal OpenFlow network and would lead to more than 90% of events being processed locally under reasonable assumptions, as described in Section 3. Applications of Kandoo are not limited to the evaluation scenarios presented in this paper. In Section 4, we briefly discuss other potential applications of Kandoo and compare it to existing solutions. We conclude our discussion in Section 5.

2. DESIGN AND IMPLEMENTATION

Design objectives. Kandoo is designed with the following goals in mind. First, Kandoo must be compatible with OpenFlow: we do not introduce any new data plane functionality in switches, and, as long as they support OpenFlow, Kandoo supports them, as well. Second, Kandoo automatically distributes control applications without any manual intervention. In other words, Kandoo control

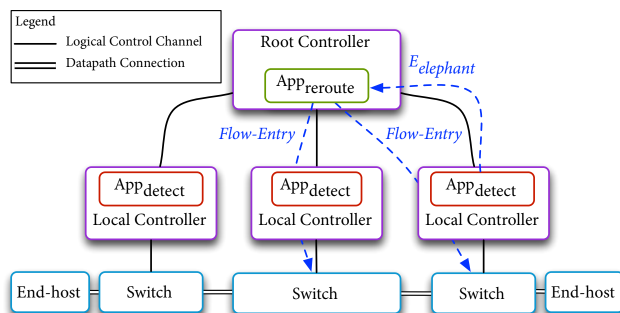


Figure 2: Toy example for Kandoo's design: In this example, two hosts are connected using a simple line topology. Each switch is controlled by one local Kandoo controller. The root controller controls the local controllers. In this example, we have two control applications: App_{detect} is a local control application, but $App_{reroute}$ is non-local.

applications are not aware of how they are deployed in the network, and application developers can assume their applications would be run on a centralized OpenFlow controller. The only extra information Kandoo needs is a flag showing whether a control application is local or not.

In what follows, we explain Kandoo's design using a toy example. We show how Kandoo can be used to reroute elephant flows in a simple network of three switches (Figure 2). Our example has two applications: (i) App_{detect}, and (ii) $App_{reroute}$. App_{detect} constantly queries each switch to detect elephant flows. Once an elephant flow is detected, App_{detect} notifies $App_{reroute}$, which in turn may install or update flow-entries on network switches.

It is extremely challenging, if not impossible, to implement this application in current OpenFlow networks without modifying switches [5]. If switches are not modified, a (logically) centralized control needs to frequently query all switches, which would place a considerable load on control channels.

Kandoo Controller. As shown in Figure 3, Kandoo has a *controller* component at its core. This component has the same role as a general OpenFlow controller, but it has Kandoo-specific extensions for identifying application requirements, hiding the complexity of the underlying distributed application model, and propagating events in the network.

A network controlled by Kandoo has multiple *local controllers* and a logically centralized *root controller*.[1] These controllers collectively form Kandoo's distributed control plane. Each switch is controlled by only one Kandoo controller, and each Kandoo controller can control multiple switches. If the root controller needs to install flow-entries on switches of a local controller, it delegates the requests to the respective local controller. Note that for high availability, the root controller can register itself as the slave controller for a specific switch (this behavior is supported in OpenFlow 1.2 [1]).

[1]We note that the root controller in Kandoo can itself be logically/physically distributed. In fact, it is straightforward to implement Kandoo's root controller using Onix [8] or Hyperflow [18].

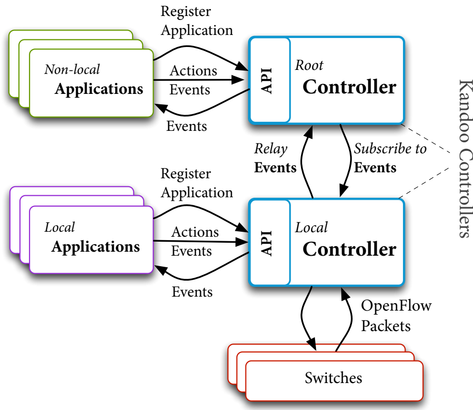


Figure 3: Kandoo's high level architecture.

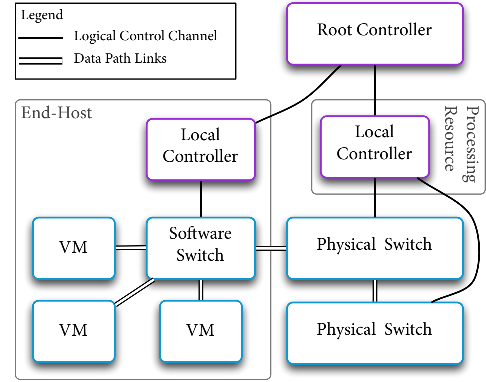


Figure 4: Kandoo in a virtualized environment. For software switches, we can leverage the same end-host for local controllers, and, for physical switches, we use separate processing resources.

Deployment Model. The deployment model of Kandoo controllers depends on the characteristics of a network. For software switches, local controllers can be directly deployed on the same end-host. Similarly, if we can change the software of a physical switch, we can deploy Kandoo directly on the switch. Otherwise, we deploy Kandoo local controllers on the processing resources closest to the switches. In such a setting, one should provision the number of local controllers based on the workload and available processing resources. Note that we can use a hybrid model in real settings. For instance, consider a virtualized deployment environment depicted in Figure 4, where virtual machines are connected to the network using software switches. In this environment, we can place local controllers in end-hosts next to software switches and in separate nodes for other switches.

In our toy example (Figure 2), we have four Kandoo controllers: three local controllers controlling the switches and a root controller. The local controllers can be physically positioned using any deployment model explained above. Note that, in this example, we have the *maximum* number of local controllers required.

Control Applications. Control applications function using the abstraction provided by the controller and are not aware of Kandoo internals. They are generally OpenFlow applications and can therefore send OpenFlow messages and listen on events. Moreover, they can *emit* Kandoo events (*i.e.*, internal events), which can be consumed by other applications, and they can *reply* to the application that emitted an event. Control applications are loaded in local name spaces and can communicate using only Kandoo events. This is to ensure that Kandoo does not introduce faults by offloading applications.

In our example, $E_{elephant}$ is a Kandoo event that carries matching information about the detected elephant flow (*e.g.*, its OpenFlow match structure) and is emitted by App_{detect}.

A local controller can run an application only if the application is local. In our example, $App_{reroute}$ is not local, *i.e.*, it may install flow-entires on any switch in the network. Thus, the root controller is the only controller able to run $App_{reroute}$. In contrast, App_{detect} is local; therefore, all controllers can run it.

Event Propagation. The root controller can subscribe to specific events in the local controllers using a simple messaging channel plus a filtering component. Once the local controller receives and locally processes an event, it relays the event to the root controller for further processing. Note that all communications between Kandoo controllers are event-based and asynchronous.

In our example, the root controller subscribes to events of type $E_{elephant}$ in the local controllers since it is running $App_{reroute}$ listening on $E_{elephant}$. $E_{elephant}$ is fired by an App_{detect} instance deployed on one of the local controllers and is relayed to root controller. Note that if the root controller does not subscribe to $E_{elephant}$, the local controllers will not relay $E_{elephant}$ events.

It is important to note that the data flow in Kandoo is not always bottom-up. A local application can explicitly request data from an application deployed on the root controller by emitting an event, and applications on the root controllers can send data by replying to that event. For instance, we can have a topology service running on the root controller that sends topology information to local applications by replying to events of a specific type.

Reactive vs. Proactive. Although Kandoo provides a scalable method for event handling, we strongly recommend pushing network state proactively. We envision Kandoo to be used as a scalable, adaptive control plane, where the default configuration is pushed proactively and is adaptively refined afterwards. In our toy example, default paths can be pushed proactively, while elephant flows will be rerouted adaptively.

Implementation Details. We implemented Kandoo in a mixture of C, C++, and Python. Our implementation has a low memory footprint and supports dynamically loadable plug-ins, which can be implemented in C, Python, and Java. It also provides an RPC API for more general integration scenarios. Our implementation of Kandoo is extremely modular; any component or back-end can be easily replaced, which simplifies porting Kandoo to physical switches. Currently, Kandoo supports OpenFlow 1.0 (OpenFlow 1.1 and 1.2 support is under development).

For the applications, we created a "central application repository" and developed a simple package management

system, which supports application dependency. When booting up, a controller downloads the information about all applications from the repository and then retrieves the applications it can run.

Single-node Performance. Single-node performance of Kandoo is of significant importance since local controllers process frequent events from single switches. For this reason, our implementation is efficient and has low overhead. A single Kandoo controller can reach a throughput of more than 1M pkt-in per second from 512 switches using a single thread on a Xeon E7-4807.

3. EVALUATION

We have evaluated Kandoo using different applications in an emulated environment. For the sake of space, we present only the results obtained for the elephant flow detection problem. This evaluation demonstrates the feasibility of Kandoo to distribute the control plane at scale and strongly supports our argument.

Setup. In our experiments, we realize a two-layer hierarchy of Kandoo controllers as shown in Figure 5. In each experiment, we emulate an OpenFlow network using a slightly modified version of Mininet [9] hosted on a physical server equipped with 64G of RAM and 4 Intel Xeon(R) E7-4807 CPUs (each with 6 cores). We use OpenVSwitch 1.4 [11] as our kernel-level software switch.

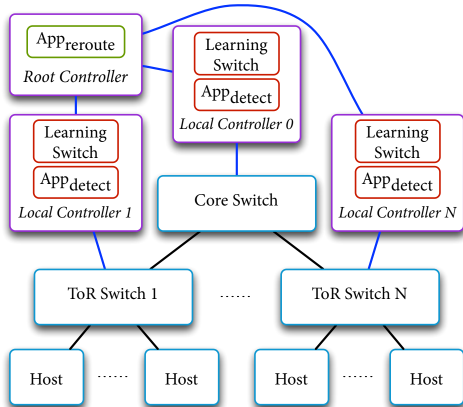


Figure 5: Experiment Setup. We use a simple tree topology. Each switch is controlled by one local Kandoo controller. The root Kandoo controller manages the local controllers.

Elephant Flow Detection. We implemented Elephant Flow Detection applications (*i.e.*, $App_{reroute}$ and App_{detect}) as described in Section 2. As depicted in Figure 5, App_{detect} is deployed on all Kandoo local controllers, whereas $App_{reroute}$ is deployed only on the root controller. Our implementation of App_{detect} queries only the top-of-rack (ToR) switches to detect the elephant flows. To distinguish ToR switches from the core switch, we implemented a simple link discovery technique. App_{detect} fires one query per flow per second and reports a flow as elephant if it has sent more than 1MB of data. Our implementation of $App_{reroute}$ installs new flow entries on all switches for the detected elephant flow.

Learning Switch. In addition to these two applications, we use a simple learning switch application on all controllers to setup paths. This application associates MAC addresses to ports and installs respective flow entries on the switch. We note that the bandwidth consumed for path setup is negligible compared to the bandwidth consumption for elephant flow detection. Thus, our evaluation results would still apply, even if we install all paths proactively.

Methodology. In our experiments, we aim to study how this control plane scales with respect to the number of elephant flows and network size compared to a normal OpenFlow network (where all three applications are on a single controller). We measure the number of requests processed by each controller and their bandwidth consumption. We note that our goal is to decrease the load on the root controller in Kandoo. Local controllers handle events locally, which consume far less bandwidth compared to events sent to the root controller. Moreover, Kandoo's design makes it easy to add local controllers when needed, effectively making the root controller the only potential bottleneck in terms of scalability.

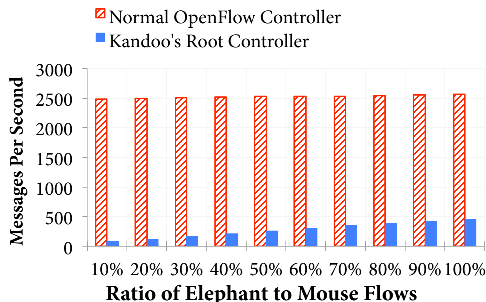
Our first experiment studies how these applications scale with respect to the number of elephant flows in the network. In this experiment, we use a tree topology of depth 2 and fanout 6 (*i.e.*, 1 core switch, 6 top-of-rack switches, and 36 end-hosts). Each end-host initiates one hundred UDP flows to any other host in the network. This synthetic workload stresses Kandoo since most flows are not local. As reported in [3], data center traffic has locality, and Kandoo would therefore perform better in practice.

As depicted in Figure 6, control channel consumption and the load on the central controller are considerably lower for Kandoo, even when all flows are elephant[2]: five times smaller in terms of packets (Figure 6(a)), and an order of magnitude in terms of bytes (Figure 6(b)). The main reason is that, unlike the normal OpenFlow, the central controller does not need to query the ToR switches. Moreover, App_{detect} fires one event for each elephant flow, which results in significantly less events.

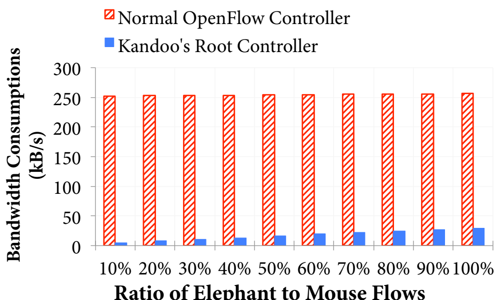
To study Kandoo's scalability based on the number of nodes in the network, we fix ratio of the elephant flows at 20% and experiment with networks of different fanouts. As illustrated in Figure 7, the role of local controllers is more pronounced when we have larger networks. These controllers scale linearly with the size of the network and effectively shield the control channels from frequent events. Consequently, Kandoo's root controller handles significantly less events compared to a normal OpenFlow network.

Based on these two experiments, Kandoo scales significantly better than a normal OpenFlow network. It is important to note that the performance of Kandoo can be improved further by simple optimization in App_{detect}. For example, the load on the central controller can be halved if App_{detect} queries only the flow-entries initiated by the hosts directly connected to the respective switch.

[2]When all flows are elephant, App_{detect} fires an event for each flow. Thus, it reflects the maximum number of events Kandoo will fire for elephant flow detection.

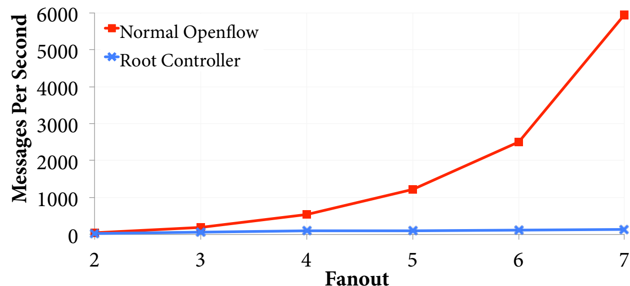


(a) Average number of messages received by the controllers

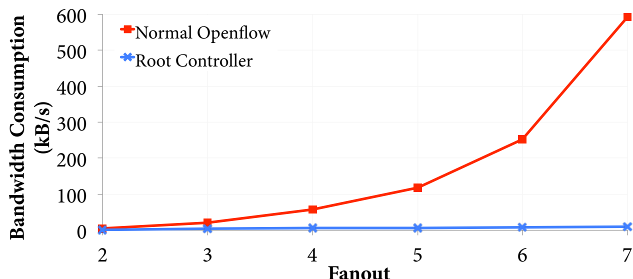


(b) Average number of bytes received by the controllers

Figure 6: Control Plane Load for the Elephant Flow Detection Scenario. The load is based on the number of elephant flows in the network.



(a) Average number of packets received by the controllers



(b) Average number of bytes received by the controllers

Figure 7: Control Plane Load for the Elephant Flow Detection Scenario. The load is based on the number of nodes in the network.

4. RELATED WORK

Datapath Extensions. The problem that we tackle in this paper is a generalization of several previous attempts at scaling SDNs. A class of solutions, such as DIFANE [21] and DevoFlow [5], address this problem by extending data plane mechanisms of switches with the objective of reducing the load towards the controller. DIFANE tries to partly offload forwarding decisions from the controller to special switches, called authority switches. Using this approach, network operators can reduce the load on the controller and the latencies of rule installation. DevoFlow, on the other hand, introduces new mechanisms in switches to dispatch far fewer "important" events to the control plane. Kandoo has the same goal, but, in contrast to DIFANE and DevoFlow, it does not extend switches; instead, it moves control plane functions closer to switches. Kandoo's approach is more general and works well in data centers, but it might have a lower throughput than specific extensions implemented in hardware.

Interestingly, we can use Kandoo to prototype and test DIFANE, DevoFlow, or other potential hardware extensions. For instance, an authority switch in DIFANE can be emulated by a local Kandoo controller that manages a subset of switches in the network. As another example, DevoFlow's extensions can also be emulated using Kandoo controllers directly installed on switches. These controllers not only replace the functionality of DIFANE or DevoFlow, but they also provide a platform to run any local control application in their context.

Distributed Controllers. HyperFlow [18], Onix [8], SiBF [10], and Devolved Controllers [17] try to distribute the control plane while maintaining logically centralized, eventually consistent network state. Although these approaches have their own merits, they impose limitations on applications they can run. This is because they assume that all applications require the network-wide state; hence, they cannot be of much help when it comes to local control applications. That said, the distributed controllers can be used to realize a scalable *root* controller, the controller that runs non-local applications in Kandoo.

Middleboxes. Middlebox architectures, such as FlowStream [7], SideCar [15] and CoMb [13], provide scalable programmability in data plane by intercepting flows using processing nodes in which network applications are deployed. Kandoo is orthogonal to these approaches in the sense that it operates in the control plane, but it provides a similar distribution for control applications. In a network equipped with FlowStream, SideCar or CoMb, Kandoo can share the processing resources with middleboxes (given that control and data plane applications are isolated) in order to increase resource utilization and decrease the number of nodes used by Kandoo.

Active Networks. Active Networks (AN) and SDNs represent different schools of thought on programmable networks. SDNs provide programmable control planes, whereas ANs allow programmability in networking elements at packet

transport granularity by running code encapsulated in the packet [20, 12, 19] or installed on the switches [16, 6]. An extreme deployment of Kandoo can deploy local controllers on all switches in the network. In such a setting, we can emulate most functionality of ANs. That said, Kandoo differs from active networks in two ways. First, Kandoo does not provide in-bound packet processing; instead, it follows the fundamentally different approach proposed by SDNs. Second, Kandoo is not an all-or-nothing solution (*i.e.*, there is no need to have Kandoo support on switches). Using Kandoo, network operators can still gain efficiency and scalability using commodity middle boxes, each controlling a partition of switches.

5. CONCLUSION

Kandoo is a highly configurable and scalable control plane. It uses a simple yet effective approach for creating a distributed control plane: it processes frequent events in highly replicated local control applications and rare events in a central location. As confirmed by our experiments, Kandoo scales remarkably better than a normal OpenFlow implementation, without modifying switches or using sampling techniques.

Kandoo can co-exist with other controllers by using either FlowVisor [14] or customized Kandoo adapters. Having said that, extra measures should be taken to ensure consistency. The major issue is that Kandoo local controllers do not propagate an OpenFlow event unless the root controller subscribes to that event. Thus, without subscribing to all OpenFlow events in all local controllers, we cannot guarantee that existing OpenFlow applications work as expected.

Moving forward, we are extending Kandoo to support new categories of control applications that are not necessarily local but that have a limited scope. Such applications can operate by having access to the events generated by a subset of switches. Using a hierarchy of controllers (as opposed to the two-level hierarchy presented in this paper), we can extend Kandoo to run such applications. Finally, we have started porting Kandoo to physical switches. Switches equipped with Kandoo can natively run local control applications.

6. REFERENCES

[1] OpenFlow Switch Specification, Version 1.2 (Wire Protocol 0x03). `http://tinyurl.com/84kelcj`.

[2] Station and Media Access Control Connectivity Discovery, IEEE Standard 802.1AB. `http://tinyurl.com/6s739pe`.

[3] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 conference*, pages 63–74, 2010.

[4] A. R. Curtis, W. Kim, and P. Yalagandula. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *Proceedings of the IEEE INFOCOM 2011 conference*, pages 1629–1637, 2011.

[5] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 conference*, pages 254–265, 2011.

[6] D. Decasper and B. Plattner. DAN: distributed code caching for active networks. In *Proceedings of the IEEE INFOCOM'98 conference*, volume 2, pages 609–616, 1998.

[7] A. Greenhalgh, F. Huici, M. Hoerdt, P. Papadimitriou, M. Handley, and L. Mathy. Flow processing and the rise of commodity network hardware. *SIGCOMM Comput. Commun. Rev.*, 39(2):20–26, 2009.

[8] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: a distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX OSDI conference*, pages 1–6, 2010.

[9] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the ACM SIGCOMM HotNets workshop*, pages 19:1–19:6, 2010.

[10] C. A. B. Macapuna, C. E. Rothenberg, and F. Magalh. In-packet bloom filter based data center networking with distributed openflow controllers. In *Proceedings of IEEE International Workshop on Management of Emerging Networks and Services*, pages 584–588, 2010.

[11] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. Extending networking into the virtualization layer. In *Proceedings of the ACM SIGCOMM HotNets workshop*, 2009.

[12] B. Schwartz, A. W. Jackson, W. T. Strayer, W. Zhou, R. D. Rockwell, and C. Partridge. Smart packets: applying active networks to network management. *ACM Transactions on Computer Systems*, 18(1):67–88, Feb. 2000.

[13] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *Proceedings of NSDI 12*, 2012.

[14] R. Sherwood, M. Chan, A. Covington, G. Gibb, M. Flajslik, N. Handigol, T. Huang, P. Kazemian, M. Kobayashi, J. Naous, S. Seetharaman, D. Underhill, T. Yabe, K. Yap, Y. Yiakoumis, H. Zeng, G. Appenzeller, R. Johari, N. McKeown, and G. Parulkar. Carving research slices out of your production networks with OpenFlow. *SIGCOMM CCR*, 40(1):129–130, Jan 2010.

[15] A. Shieh, S. Kandula, and E. G. Sirer. Sidecar: building programmable datacenter networks without programmable switches. In *Proceedings of the ACM HotNets worksop*, pages 21:1–21:6, 2010.

[16] J. M. Smith, D. J. Farber, C. A. Gunter, S. M. Nettles, D. C. Feldmeier, and W. D. Sincoskie. SwitchWare: accelerating network evolution (White paper). Technical report, 1996.

[17] A.-W. Tam, K. Xi, and H. Chao. Use of devolved controllers in data center networks. In *Processings of the IEEE Computer Communications Workshops*, pages 596–601, 2011.

[18] A. Tootoonchian and Y. Ganjali. Hyperflow: a distributed control plane for openflow. In *Proceedings of the 2010 INM conference*, pages 3–3, 2010.

[19] D. J. Wetherall, J. V. Guttag, and D. L. Tennenhouse. ANTS: a toolkit for building and dynamically deploying network protocols. In *Proceedings of the IEEE Open Architectures and Network Programming conference*, pages 117–129, 1998.

[20] D. J. Wetherall and D. L. Tennenhouse. The ACTIVE IP option. In *Proceedings of the 7th ACM SIGOPS European workshop on systems support for worldwide applications*, pages 33–40, 1996.

[21] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with DIFANE. In *Proceedings of the ACM SIGCOMM 2010 conference*, pages 351–362, 2010.