Devolve-Redeem: Hierarchical SDN Controllers with Adaptive Offloading

Rinku Shah
Indian Institute of Technology
Bombay, India
rinku@cse.iitb.ac.in

Mythili Vutukuru
Indian Institute of Technology
Bombay, India
mythili@cse.iitb.ac.in

Purushottam Kulkarni
Indian Institute of Technology
Bombay, India
puru@cse.iitb.ac.in

ABSTRACT

Towards improving SDN control plane scalability, past work has proposed SDN controller frameworks that offload computation which depends on local state to controllers residing on the switches. Our work identifies another type of computation that can be offloaded to local controllers: that which depends on state that is generated globally but can be used within local controllers with loose synchronization. Because using such state locally incurs a synchronization cost, such offload makes sense only when the benefits of the offload out-weigh the synchronization cost. We present the design and implementation of Devolve-Redeem, an SDN controller framework that can offload computation to local controllers depending on the mix of various control messages in the incoming traffic. The offload decision in our framework is made by computing a cost metric that captures the relative costs of processing every control message at the central and local controllers, taking into account synchronization costs. The SDN application developer using our framework writes a single application that runs at both the central and local controllers, using our state management API to access offloadable state. Our framework migrates between various offload modes using the computed cost metric, by manipulating the rules in the SDN switches that forward control messages to the controllers. Our framework also transparently handles state synchronization between central and local controllers in a manner that is consistent with the offload mode. We have implemented the SDN-based LTE EPC application in our framework, and experiments with our prototype demonstrate the effectiveness of our adaptive offload framework.

APNET '17, August 3–4, 2017, Hong Kong, China
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5244-4/17/08…$15.00
https://doi.org/10.1145/3106989.3107001

CCS CONCEPTS

• **Networks → Middle boxes / network appliances**; *Programming interfaces*;

KEYWORDS

software-defined networking, scalability

ACM Reference format:
Rinku Shah, Mythili Vutukuru, and Purushottam Kulkarni. 2017. Devolve-Redeem: Hierarchical SDN Controllers with Adaptive Offloading. In *Proceedings of APNET '17, Hong Kong, China, August 3–4, 2017,* 7 pages.
https://doi.org/10.1145/3106989.3107001

1 INTRODUCTION

Software Defined Networking (SDN), a design paradigm for separating the control and data plane in networking elements, consists of a software-managed logically centralized controller and light-weight switches for data forwarding. Using its network-wide view, the controller installs rules on switches to handle packet forwarding. Any traffic for which rules do not exist, or signaling messages that require control plane processing, are directed to the controller by the switches. Prior research (§4) has identified several scalability problems with centralized SDN controllers, and with the communication path between the data plane switches and controllers, and has proposed solutions to fix the same. One set of solutions [6, 8] develop *horizontally* scalable SDN controllers that spawn multiple replicas with increasing load. Other solutions [3, 4, 9] propose *hierarchical* SDN controller frameworks which offload computation that does not require network-wide view to local controllers running on (or close to) the switches. For example, traffic engineering applications that detect flows with large number of packets (elephant flows) can maintain local state of flow statistics at the switches, and offload the task of detecting large flows to local controllers. The centralized controller can only run route computations that require global view, thereby reducing its computation load.

Our observation is that the domain of possible state and computation offloads has not been sufficiently explored. Beyond the simple taxonomy of global network-wide state and

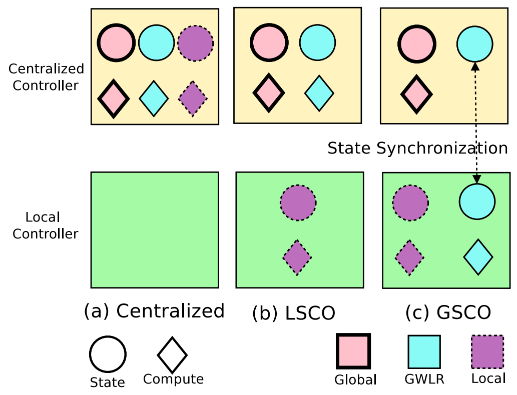

Figure 1: Design options for SDN controllers.

local switch-specific state,there is application state may depend on a global network-wide view occasionally (e.g., during creation), but can be used locally at the switches for the most part, with only loose synchronization with the centralized global state. We refer to such state as globally writeable but locally readable, or *GWLR state*. Thus all internal state in an application can be classified as GWLR state, local state (switch-specific), and global state (network-wide state that is not GWLR). While prior work has only considered offloading computations that rely on local state, we believe that computations that rely on GWLR state can also be safely offloaded to local controllers, easing the bottleneck on the control plane further. Our work proposes a new hierarchical SDN controller framework, which we call the GWLR State based Compute Offload (GSCO), where computations that depend on GWLR state are offloaded to local controllers. On the other hand, the SDN controller frameworks that only offload computation that depends on local state will be referred to as Local State based Compute Offload (LSCO) frameworks. Note that some prior research (Beehive [10]) does enable offloading computation that depends on any application state to any one of a set of distributed controllers, by moving around the state stored in a distributed data store. However, such dynamic placement of state incurs overheads like running a consensus algorithm across controllers to determine state placement, thereby impacting performance. In contrast, our work proposes offloading only application-specific state that requires loose synchronization with centralized state (and the computation that depends on it) to local controllers.

We now illustrate this key idea of our work with a motivating example. One of the network components being considered for control-data plane decomposition in telecom networks is the mobile cellular packet core, also called the LTE EPC (Long Term Evolution Evolved Packet Core). The EPC is part of the 4G LTE network that connects the wireless side of the network (the user and the base stations) to the rest of the Internet (refer Figure 3). The main network elements in the EPC are the Mobility Management Entity (MME) in the control plane, and the Serving and Packet Gateways (SGW and PGW) that forward user traffic in the data plane. Recent proposals to redesign the EPC using SDN principles [2] propose decomposing the control and data plane logic in the EPC gateways, and running the control parts of the gateways along with the MME in an SDN controller.

When a user equipment (UE) connects to a LTE network for the first time, the MME processes a control plane *attach* request to register the user, and sets up corresponding forwarding state for the users in the SGW and PGW. When the UE becomes active after an idle period, it generates a *service request* to restore the previous forwarding state that was released in the idle period. In a traditional SDN architecture of the EPC, all signaling messages, including the attach request and service request, are forwarded to the centralized controller. Now, most of the processing of an attach request must necessarily be done in the centralized controller because it requires global state to authenticate the user and create forwarding state that relies on the global network topology. However, processing the service request requires only the forwarding state that was already created during the attach procedure, and can be entirely offloaded to a local controller, provided the globally created forwarding state is made available locally. A GSCO SDN controller framework with local controllers at EPC gateway switches can thus efficiently offload computations like service request processing.

While the GSCO controller design decreases load on the central controller, it also incurs the cost of synchronizing GWLR state from the centralized to the local controllers. Therefore, in traffic mixes that require very little computation on GWLR state, this synchronization cost outweighs the benefit of compute offload, and a traditional centralized controller design might work better. The LSCO controller design offloads only a subset of computation as compared to the GSCO design, but does not carry any synchronization costs. Therefore, for any networking application, the right amount of offload that is optimal (Centralized or LSCO or GSCO) depends on the traffic characteristics, among other things. With traffic characteristics being dynamic in nature, an SDN controller framework that supports offloading must adaptively manage the amount of traffic offloaded to local controllers in order to optimize overall system performance. Figure 1 illustrates the various offload design options described here. Each offload mode (Centralized, LSCO, GSCO) differs in the placement of the global, local, and GWLR state, and hence in the computation that works on such state.

This paper proposes Devolve-Redeem, a hierarchical SDN controller framework that adaptively offloads computation to local controllers on dataplane switches to improve SDN control plane scalability (§2). SDN application developers

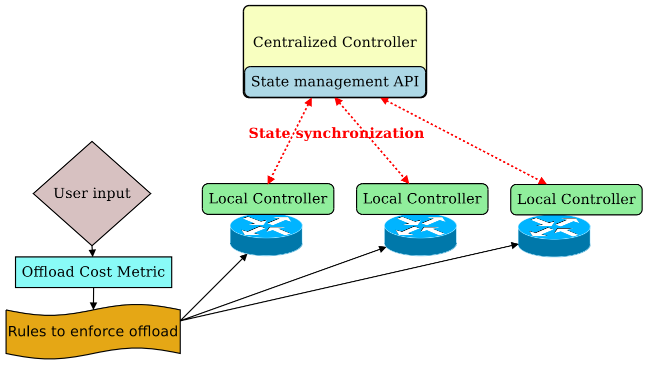

Figure 2: Devolve-Redeem Architecture.

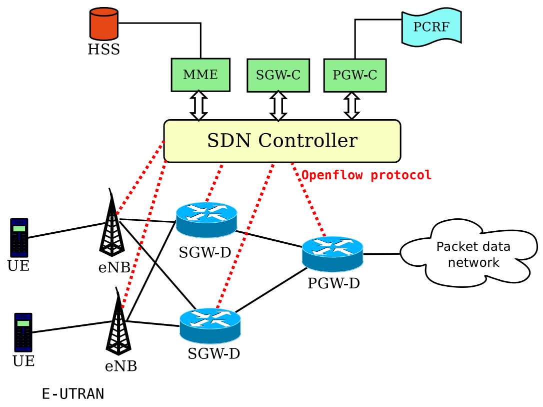

Figure 3: SDN-based LTE EPC Architecture.

using our framework only write a single application that runs both at the central and local controllers. The application developer provides our framework with inputs about the different control plane messages in the traffic that require processing at the SDN controller, and information about the kind of state (global, local, GWLR) accessed during the processing of each message. With this input, our framework computes the relative costs of processing each message at the central controller and at the local controller. Using these costs, and information about the traffic mix, our framework computes the optimal operating mode (Centralized, LSCO, GSCO) for the SDN controller. The framework decides a suitable controller (central or local) that is best suited to process each type of message, and pushes rules into the SDN switches to enforce this decision; we assume that control message types can be identified from packet headers by SDN switches. We also provide a state management API to write applications, which automatically synchronizes GWLR state at the central controller with local controllers when in GSCO mode, enabling state synchronization in a manner that is transparent to the application developer.

We evaluate our adaptive hierarchical SDN controller framework using the SDN-based LTE EPC as the example SDN application (§3) to highlight the effectiveness of the LSCO and GSCO modes, and the need to adapt the offload mode based on the traffic mix.

2 DESIGN AND IMPLEMENTATION

We now present the design of our Devolve-Redeem SDN controller framework, the main components of which are shown in Figure 2. We begin with a description of the LTE EPC control plane application used as an example to illustrate our system (§2.1). We then describe the user inputs required and the API available to application developers (§2.2), followed by a discussion on the cost metric that decides the offload mode suitable for a given traffic mix (§2.3). Finally, we describe the changes to the SDN controllers and switches required to enforce our state and compute placement in accordance with the selected offload mode (§2.4).

2.1 LTE EPC Control Plane Application

Towards a flexible and scalable LTE packet core, several researchers have proposed a redesign based on SDN principles [2]. We use this application as the running example to illustrate our system, primarily because of the rich complexity of the application, and the large amount of control and signaling messages involved (that is typical of telecom architectures, in contrast with enterprise IP networks). Figure 3 shows the SDN-based architecture of the LTE EPC, where components like the Mobility Management Entity (MME), Home Subscriber Server (HSS), Policy and Charging Rules Function (PCRF) that deal with control message processing are implemented as SDN applications in an SDN controller. In addition, the Serving and Packet Gateways (SGW and PGW) that forward user traffic through the mobile core are decomposed to move their control sub-components to the SDN controller. In this architecture, control plane messages are redirected to and processed at the SDN controller, and rules to forward user data are installed in the LTE gateways, which are built as SDN switches. In our paper, we primarily deal with two types of control messages: the *attach* request that is sent when a user wishes to initiate a mobile data connection, and a *service* request that is sent after an idle period to reactivate the data connection. While our design extends to other types of control messages, we restrict our discussion to these message types for ease of exposition.

Our implementation of Devolve-Redeem extends an open-source SDN-based EPC code base [5], which uses the Floodlight SDN controller and Open vSwitch (OVS) SDN switches. A multithreaded load generator (part of the original code base) simulates multiple concurrent users generating various control and data messages to the packet core. While the initial implementation represents a centralized SDN architecture, we modified it to implement the LSCO and GSCO modes. We have extended the load generator to tag packets with message types in the IP ToS field, as required by our framework. We also made changes to the LTE EPC application to use our state management API.

2.2 User inputs and API

User input forms an important aspect of the design of Devolve-Redeem, and distinguishes it significantly from prior work. We use hints about the application and the nature of computation of various control plane messages to make optimal offload decisions. We assume that the control plane traffic to the application (which is of interest to us, given that we are dealing with an SDN controller framework) has a discrete, known number of message types, which can be identified by inspecting packets in the SDN switches. Note that the attach and service requests are composed of several messages. The application programmer provides to our framework input in the form of tuples: (Msg-id, <N^{LR}, N^{LW}, N^{XR}, N^{XW}, N^{GR}, N^{GW}>, N^R) Msg-id is the unique identifier for each message type. N^R denotes the number of rules installed/removed/modified on the switch, by the processing of a certain message. N^K denotes the number of units of type K, where K is one of: LR—Local Read, LW—Local Write, XR—GWLR Read, XW—GWLR Write, GR—Global Read, GW—Global Write. We expect that application developers will have sufficient knowledge about the state access patterns of their application to be able to provide such an input. A sample user input for Devolve-Redeem is shown in Table 1. This table shows the control message types that are exchanged during the attach and service request processing, and their corresponding input tuple

Message type	**Input tuple**
Auth_Step_1	$1, < 0, 0, 0, 0, 1, 2 >, 0$
Auth_Step_3	$2, < 0, 0, 0, 0, 3, 2 >, 0$
NAS_Step_2	$3, < 0, 0, 0, 0, 1, 0 >, 0$
Send_APN	$4, < 0, 0, 0, 5, 1, 1 >, 4$
Send_UE_TEID	$5, < 0, 0, 2, 1, 0, 0 >, 2$
UE Context Release	$6, < 2, 0, 0, 1, 0, 0 >, 3$
UE Service Req	$7, < 0, 0, 0, 1, 0, 0 >, 1$
Context Setup Resp	$8, < 0, 0, 1, 1, 0, 0 >, 2$

Table 1: Sample user input.

In addition to providing input about the various control messages, developers must also use our state management API to access control plane state. We assume that all GWLR state is stored as key-value pairs, and the key can be used to identify the local switch/controller to which that state belongs. Our API provides get/put/delete functions, which are invoked when accessing the GWLR state in the application code. The state management logic in the controllers is aware of the offload mode being used, and synchronizes GWLR state across the central and local controllers using Floodlight's synchronization service [7], only when functioning in GSCO mode. As a result, the application developer does not have to write separate applications at the central and local controllers, and for different offload modes. Note that our framework does not maintain any versioning of the state, and does not guard against application errors that happen due to reading stale state. We expect that the application code will track the freshness of the state using techniques like version numbers that can be updated by both the centralized and local controllers. For example, if the application finds from the version number that the state has not yet been synchronized from the global controller, it must wait suitably till the desired state is updated.

2.3 Offload cost metric

We now describe a metric that computes the cost of processing a certain message type in a certain offload mode, based on user input provided in §2.2. We then compute the weighted cost of a traffic mix for each offload mode by taking a weighted average of the costs of the constituent messages in the mix. By comparing the relative costs of handling a traffic mix in various offload modes, our framework can decide on the optimal offload configuration for a given traffic mix. In our current design, this cost computation is done offline to choose an offload mode, but our design is easily amenable to dynamically computing this cost (§5).

In addition to the user provided input, we use the following parameters obtained from empirical measurements: C^K is the cost of accessing state of type K from a switch (K is one of LR, LW, XR, XW, GR, GW), D^C is the communication cost between the central controller and the switch, D^L is the communication cost between the local controller and the switch , and C^S is the cost of synchronizing GWLR state across controllers. All costs are in relative units to denote processing/network delays or any other cost. $\exists(x)$ is a boolean variable that is 1 if $x > 0$ and 0 otherwise. As part of future work, we are working on how the framework can derive these parameters automatically.

The cost of processing a message in the centralized mode consists of the cost of accessing all state from the central controller, the RTT of communicating the message from the switch to the controller, and the cost of installing rules from the central controller at the switch, as shown below.

$$\begin{aligned} C^{CENT} = (N^{LR} + N^{XR} + N^{GR}) * C^{GR} + \\ (N^{LW} + N^{XW} + N^{GW}) * C^{GW} + 2 * D^C + (N^R * D^C) \end{aligned} \tag{1}$$

LTE Procedure	C_{proc}^{CENT}	C_{proc}^{LSCO}	C_{proc}^{GSCO}
Attach	31.44	31.44	37.11
Service Request	39.56	34.79	25.64

Table 2: Relative cost of attach and service request procedures across all offload modes.

The cost of processing a message in the LSCO mode will account for the fact that local state accesses incur a different cost from global/GWLR state accesses, and processing of a message requires communication with the central controller only if the message accesses any global/GWLR state ('|' indicates boolean OR).

$$\begin{aligned} C^{LSCO} &= (N^{LR} * C^{LR}) + (N^{LW} * C^{LW}) + \\ &(N^{XR} + N^{GR}) * C^{GR} + (N^{XW} + N^{GW}) * C^{GW} + \\ &(N^{XR} \mid N^{XW} \mid N^{GR} \mid N^{GW}) * 2 * D^{C} + (N^{R} * D^{L}) \end{aligned} \tag{2}$$

The cost of processing a message in GSCO mode will account for accessing both local and GWLR state locally, and the cost of synchronizing GWLR state.

$$\begin{aligned} C^{GSCO} &= (N^{LR} + N^{XR}) * C^{LR} + (N^{LW} + N^{XW}) * C^{LW} + \\ &(N^{GR} * C^{GR}) + (N^{GW} * C^{GW}) + (N^{GR} \mid N^{GW}) * 2 * D^{C} + \\ &(\exists(N^{XR}) + N^{XW}) * C^{S} + (N^{R} * D^{L/C}) \end{aligned} \tag{3}$$

Finally, we compute the weighted cost of processing a traffic mix in a given mode as a weighted sum of, the relative fraction of that message type in the traffic (w_i), the relative state access cost (C_i^{mode}) computed above, and the processing costs of the messages in the procedure (CPU_i). The CPU_i values would be dynamically computed by our framework.

$$C_{mix}^{mode} = \sum_i w_i * C_i^{mode} * CPU_i \tag{4}$$

We now illustrate our cost computation using the EPC control plane example, with a mix of attach and service request procedures. We use input of the form shown in Table 1 for all the control messages that are exchanged during these procedures. We then compute the cost of processing the various message types, and sum them up to compute the cost of the attach and service request procedures in various offload modes, as shown in Table 2. Finally, we use Equation 4 to compute the cost for various mixes of attach and service request procedures in the control traffic, the results of which are shown in Table 3. Our empirical results in §3 agree with our analytical computation of the best offload mode for each traffic mix.

Attach:Service-Req (x:y)	$C_{x:y}^{CENT}$	$C_{x:y}^{LSCO}$	$C_{x:y}^{GSCO}$
10:90	12.13	10.95	**9.07**
20:80	13.31	12.25	**11.04**
40:60	15.66	14.86	14.98
60:40	18	**17.48**	18.92
80:20	20.35	**20.09**	22.86
100:0	**22.70**	**22.70**	26.80

Table 3: Cost of traffic mixes across all offload modes.

2.4 Enforcing the offload mode

Once the cost metric is used to decide the offload mode and hence the optimal location to handle a certain type of message, our framework enforces this decision via OVS rules installed at all switches to direct a message to a suitable (central/local) controller. The Floodlight controller in our implementation did not implement support to direct packets to a specified controller. Therefore, we developed an extension to the Floodlight controller by implementing the "NiciraSet-ControllerId" feature in the Loxigen library, which lets us specify rules to direct specific message types to specific controllers. It was not required to make any additions to the OVS code in the switches as the "NiciraSetControllerId" feature was already supported. For each incoming control message packet at the ingress SDN switch, the message type is inspected and the control message is directed to and processed at the appropriate (centralized or local) controller.

3 EVALUATION

We now show the effectiveness of our Devolve-Redeem framework with experiments using the EPC application.

3.1 Setup

We deployed the LTE EPC control plane application described in §2.1 over our testbed. The Floodlight v1.2 controller acts as a central controller and serves 3 sets of EPC gateways along with a load generator and sink, to represent a realistic scenario where multiple data plane switches are served by the same controller. The EPC gateways run as SDN switches on OVS v2.3.2. The SGW dataplane switch also hosts a local controller. All components (controller and switches) run Ubuntu 14.04, and are hosted over separate LXC containers to ensure isolation. The containers are distributed amongst two Intel Xeon E312xx @2.6Ghz servers, one with 16 CPU cores and the other with 8 cores. The controller is allocated 1 CPU core and 4GB RAM, whereas the gateways are allocated 2 cores and 4GB RAM each. The centralized controller is allocated fewer resources in comparison to the switches to reflect real life where an MME controller could be servings tens or hundreds of EPC gateways. Our load generator generates traffic consisting of *attach*, *detach*, and *service request* messages, separated by random delays, to enforce a specific mix of attach and service request traffic. All the experiments were carried out for the duration of 300 seconds. The metrics measured are average throughput (number of requests completed/sec), average request completion latency, CPU utilization at all components, and network traffic to the central controller. The throughput and latency parameters are measured at the load generator, while the CPU and network parameters are measured at the host and the OVS switches.

3.2 Results

Figure 4 shows the average control plane throughput of various offload modes with varying mix of attach and service request messages in the traffic, e.g., 10:90 represents 10% attach requests and 90% service requests. The load generator generated the specified fraction of requests per procedure type every second using a uniform distribution. We conclude from the figure that when the ratio of attach requests is much higher than service requests, the centralized and LSCO modes worked similarly, and gave higher throughput than the GSCO mode, because there was very little GWLR computation to offload. On the other hand, when the ratio of service requests was higher, more GWLR computation could be offloaded, and the synchronization cost of GWLR state was offset by the benefit of computation offload, resulting in higher throughput for the GSCO and LSCO modes. For example, GSCO and LSCO provide 27% and 19% higher throughput respectively, as compared to centralized, when the input traffic mix is 20:80. LSCO provides 22% improvement over centralized at a mix 40:60. The performance of the GSCO design degraded when the percentage of attach requests crossed 40%.

The latency measurements showed that the Centralized mode always suffered high average request completion latency, since the packets always travel to the centralized controller, and the values range from 10.5 ms (10:90) to 15 ms (60:40). The average request completion latency in LSCO mode was lower than the Centralized mode, since certain packets were serviced at the switch, and the values range from 9 ms (10:90) to 12 ms (60:40). The average request completion latency in GSCO was lower than Centralized and LSCO mode upto the 20:80 mix (8 ms), but increased thereafter upto 14 ms for 60:40 mix, due to synchronization overheads.

Table 4 shows the traffic received at the centralized controller and the normalized CPU utilization at the local controllers with varying traffic mix. We conclude that when the ratio of attach requests was much higher than service

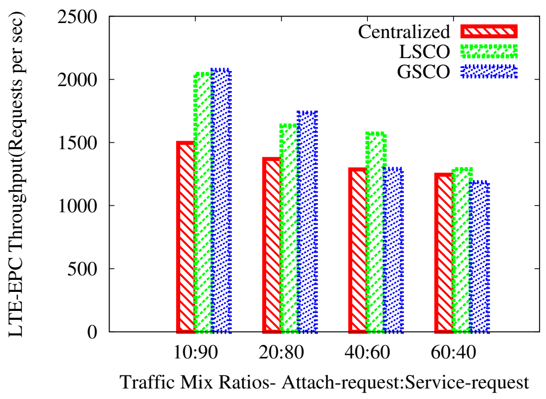

Figure 4: Control throughput in offload modes as a function of traffic mix.

Mix	Network Overhead (Mbps)			%CPU utilization at switch		
$x : y$	CENT	LSCO	GSCO	CENT	LSCO	GSCO
10:90	21	20.4	11.7	33.5	43	53.2
20:80	19.2	17.1	14.6	33.7	40.9	53.7
40:60	17.5	17.6	17.3	34.5	47.3	47.1
60:40	16.7	15.3	*21.4*	35.8	41.2	45.5

Table 4: Network overhead and CPU utilization at the switch across offload modes.

requests, GSCO imposes a significant synchronization overhead. For example, GSCO had a 28% and 40% higher network overhead compared to the centralized and LSCO designs respectively at a mix of 60:40. We also see that the LSCO and GSCO modes impose a higher CPU overhead at the local switches as compared to the centralized mode, as expected.

4 RELATED WORK

Towards realizing the applicability of software-defined networks, an important problem is that of control-plane scalability. Horizontal scaling approaches [6, 8] scale the controller by instantiating multiple homogeneous instances of the centralized controller, and distributing the control load with techniques like network topology partitioning. On the other hand, hierarchical scaling techniques such as ours scale controllers by offloading computation from the centralized controller. We now compare our work to other hierarchical controller frameworks.

Static state and computation offload: With a hierarchical SDN controller solution, one or both of computation and state are distributed across the central root controller and decentralized local controllers. Difane [11] offloads forwarding rules across a subset of local switches which serve as local controllers for new flows with no forwarding rules. The availability of rules locally (state offload) avoids expensive per-flow message transfers to the central controller. Other approaches like Kandoo [4], FOCUS [9] offload computation tasks like statistics gathering for elephant flow detection, or node discovery via ARP flooding. Most of these optimizations are related to computation tasks that do not affect global controller state. Our proposed approach not only offloads local computation, but also computation that depends on global state that can be used locally in a read only fashion.

Eden [1] provides a framework for implementing the network functions that do not require high network support at the end hosts. They observe that a network function handles as a set of application messages, and provide an application library that tags packets according to the message type. The tagged packets are then processed at the end hosts via a set of match-action tables and a runtime. Devolve-Redeem also works by identifying application message types. However, while Eden requires changes to applications and the host software stack to offload computation to the end hosts, our framework does not require any significant changes to the applications, and offloads computation to switches and not the end hosts.

Adaptive offloading of state and computation: The hierarchical controller frameworks described so far perform a static offloading of state and/or computation to local switches or end hosts, with the amount of computation to be offloaded decided at design time. Beehive [10], on the other hand, enables dynamic placement of computation by enabling the dynamic placement and movement of state across a set of distributed controllers. Application state in Beehive is stored as key-value pairs in a shared distributed data store. Computations in the application are *mapped* to the relevant controller replica using an application-specific map function that queries a globally synchronized index and maps compute tasks to wherever data resides. Beehive requires writing network applications to integrate the mapping functionality (provided by the Go language). Further, the various Beehive controllers must run an expensive synchronization protocol to agree on the location of the distributed state. As compared to Beehive, our approach trades off generality in favor of performance: Devolve-Redeem does a more restricted placement of data at local and centralized controllers, using hints from the developer, unlike the generic distributed datastore of Beehive. Further, our approach routes messgaes to controllers by identifying message types from packet headers at the switches itself, while Beehive routes packets to controllers using the map function at the application layer.

5 DISCUSSION AND CONCLUSION

We presented the design and implementation of Devolve-Redeem, a hierarchical SDN controller that can adaptively change the amount of computation offloaded to local controllers. Experiments with our implementation show that the optimal amount of offload that gives the best application performance depends on the mix of traffic in the network. Offloading computation that depends on GWLR state incurs a synchronization cost, and the decision of whether to offload computation that depends on such state, or whether to only offload computation that depends on local state (or even not offload anything at all), depends on whether the traffic has enough control message that can benefit from such an offload. We proposed a simple cost metric that can help us decide the relative costs of processing a message with and without offload, and hence the optimal offload design for a given traffic mix. Our current system performs this computation offline. However, our design has all the ingredients in place to adapt the offload mode in an online fashion, because the offload is accomplished via two mechanisms (rules at switches to redirect messages to the appropriate controller based on their message type, and state synchronization across controllers via our API) in a manner that is transparent to the application developer. Therefore, applications need not change with offload modes. A complete implementation and evaluation of a hierarchical controller that can dynamically adapt the amount of computation offloaded based on the traffic characteristics, and even the resource utilization at controllers, is part of our ongoing work.

REFERENCES

[1] Hitesh Ballani and others. Enabling End-host Network Functions. In *Proc of the SIGCOMM, 2015.*

[2] Arsany Basta and others. A Virtual SDN-Enabled LTE EPC Architecture: A Case Study for S-/P-Gateways Functions. In *Proc of IEEE SDN4FNS, 2013.*

[3] Andrew R. Curtis and others. DevoFlow: Scaling Flow Management for High-performance Networks. In *Proc of the SIGCOMM, 2011.*

[4] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. In *Proc of the Workshop on HoTSDN, 2012.*

[5] Aman Jain, Sunny Lohani, and Mythili Vutukuru. 2016. Opensource SDN LTE EPC. https://github.com/networkedsystemsIITB/SDN_LTE_EPC. (2016).

[6] Teemu Koponen and others. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proc of the Conference on OSDI, 2010.*

[7] Tulio Ribeiro. 2016. Floodlight. https://github.com/floodlight. (2016).

[8] Amin Tootoonchian and Yashar Ganjali. HyperFlow: A Distributed Control Plane for OpenFlow. In *Proc of the Internet Network Management Conference on Research on Enterprise Networking, 2010.*

[9] Ji Yang and others. FOCUS: Function Offloading from a Controller to Utilize Switch Power. In *Proc of IEEE Conference on NFV-SDN, 2016.*

[10] Soheil Hassas Yeganeh and Yashar Ganjali. Beehive: Simple Distributed Programming in Software-Defined Networks. In *Proc of the SOSR, 2016.*

[11] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. Scalable Flow-based Networking with DIFANE. In *Proc of the SIGCOMM, 2010.*