Pratyaastha: An Efficient Elastic Distributed SDN Control Plane

Anand Krishnamurthy, Shoban P. Chandrabose, Aaron Gember-Jacobson
University of Wisconsin–Madison
{anand,shoban,agember}@cs.wisc.edu

ABSTRACT

Several distributed SDN controller architectures have been proposed to mitigate the risks of overload and failure. However, since they statically assign switches to controller instances and store state in distributed data stores (which doubles flow setup latency), they hinder operators' ability to minimize both flow setup latency and controller resource consumption. To address this, we propose a novel approach for assigning SDN switches and partitions of SDN application state to distributed controller instances. We present a new way to partition SDN application state that considers the dependencies between application state and SDN switches. We then formally model the assignment problem as a variant of multi-dimensional bin packing and propose a practical heuristic to solve the problem with strict time constraints. Our preliminary evaluations show that our approach yields a 44% decrease in flow setup latency and a 42% reduction in controller operating costs.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: [Network architecture and Design]; C.2.4 [**Computer-Communication Networks**]: [Distributed Systems]

General Terms

Design, Experimentation

Keywords

software defined network controllers

1. INTRODUCTION

Software-defined networking (SDN) centralizes the network control plane [5, 10], thus enabling (optimal) forwarding decisions to be made on the basis of a global network view. However, a single centralized controller can easily succumb to overload or failure. Even a very powerful controller will lack the CPU and memory capacity necessary to maintain complete network state, and react to all network events, for large, high volume networks.

Several "logically centralized" but "physically distributed" controller architectures [6, 8, 9, 11, 13] have been proposed to address this issue. In these architectures, each controller instance is responsible for the events and actions pertaining to a subset of the network's switches.

However, current distributed controller architectures fail to adequately address two important concerns:
(*1*) Minimizing flow setup latency. Distributed controller architectures that use a static assignment of switches to controllers [8, 9, 11, 13] are highly susceptible to overload, and increased flow setup latency, because of their inability to adapt to shifts in traffic load. Furthermore, architectures that depend on distributed data stores (or remote procedure calls (RPCs)) [6, 9, 11] can consistently incur $2\times$ higher flow setup latencies (Section 2). This overhead may be acceptable when forwarding rules are proactively installed, but it can cripple many SDN applications that depend on fast reactive rule installation for correction operation: e.g., consistent load balancing [15], traffic engineering [14], and dynamic traffic filtering [2].
(*2*) Minimizing controller operating costs through efficient resource allocation. Over-provisioning controller resources to accommodate peak load can help reduce spikes in flow setup latency, but this wastes resources and increases operating costs. Dynamic resource allocation and re-balancing of controller responsibilities [6] is preferred, but existing architectures only consider CPU load (or the rate of switch events, which is usually correlated with CPU load) when assigning switches to controller instances. They do not consider memory—which is also a critical resource when application state is stored at controller instances [8, 13]—or inter-controller communication costs—e.g., a remote procedure call to install forwarding rules in a switch connected to a different controller instance consumes network bandwidth and introduces latency overhead.

To address these issues, we propose a novel approach for assigning SDN switches and partitions of SDN application state to distributed controller instances. We first present a new way to partition application state that considers the dependencies between state and switches: e.g., two controller instances each handling one endpoint of a tunnel will both depend on application state related to that tunnel [9]. We then formally model the assignment problem as a variant of multi-dimensional bin packing. We consider both CPU and memory, and we impose additional costs when switches and the application state they depend on are assigned to different bins (i.e., controller instances). Since solving this problem is $\mathcal{NP}$-hard, we propose a practical heuristic that solves the problem within strict time constraints.

We evaluate our system, called *Pratyaastha*,[1] using a series of simulations and actual SDN applications. Our results show that

[1]*Pratyaastha* means "elasticity" in Sanskrit.

our partitioning and assignment strategies yield a 42% reduction in operating costs and a 44% decrease in flow-setup latencies.

2. MOTIVATION

Several distributed SDN controller architectures have been proposed [6, 8, 9, 11, 13] to reduce the likelihood of controller overload and minimize the impact of controller failures. These architectures run multiple controller instances, with each instance handling the events (e.g., new flow, link down, etc.) and actions (e.g., install forwarding rule) pertaining to a subset of SDN switches. Each controller instance also runs a copy of each SDN application (e.g., load balancing [15], traffic engineering [14], network virtualization [9], etc.)

Switch Assignment. Switches may be assigned to controller instances based on either a static [8, 9, 11, 12, 13] or dynamic [6] *assignment strategy*. Prior work [6] has shown that a static assignment strategy suffers from both overload and inefficient resource utilization when traffic load shifts. This compromises a controller's ability to both react quickly to network events and operate with minimal resources. In contrast, dynamic assignment allows adjustments to be made in response to the volume of switch events, or current CPU load, and controller instances can be (de)allocated when additional (or less) processing capacity is required.

State Storage and Access. While processing capacity is a key factor in assigning switches to controller instances, *state storage and access* is an even more important issue. Both core controller modules (e.g., topology discovery) and SDN applications maintain state that is created and accessed while handling switch events and invoking switch control actions. Each controller instance must have access to the state required for handling both the events and actions pertaining the switches it's assigned.

A naïve approach is to replicate all state at all controller instances. This ensures the relevant state is always available regardless of which switches are assigned to an instance. However, the volume of state maintained by applications can quickly overwhelm the memory resources of controller instances [9]. Furthermore, maintaining replica consistency (if necessary) is a daunting task [13].

Therefore, most existing distributed controller architectures distribute state among controller instances [8] or use a separate distributed storage system [6, 9, 11]. However, given that accessing controller or application state lies in the critical path of handling switch events, the latency overhead imposed by a distributed storage system can unacceptably increase flow setup latency. Similar overheads can also occur when state is suboptimally distributed among controller instances—i.e., another controller instance must frequently be contacted to access state.

To quantify this latency overhead, we ran our own multi-tenant virtualized data center application, and compared the overhead of accessing state stored in a distributed data store versus storing state in memory at the controller. The application creates a full mesh of tunnels between all of a tenant's VMs and stores the details of each tunnel; upon receiving a packet for a new flow, the application fetches the tunnel data, checks if the flow is admissible, and installs forwarding rules in virtual switches at the source and destination hypervisors. We run the application in a small data center testbed (12 racks, 3 machines per rack, 3 virtual machine (VM) slots per machine) with two tenants (2 VMs per tenant). We generate ping requests between each pair of VMs and measure the RTT of the first packet for each flow. From the results in Figure 1, we see that with a distributed data store, there is an $\approx 78\%$ increase in RTT for the first packet of a flow. Such an increase can significantly impact mice flows that normally complete in just one or two RTTs. Also,

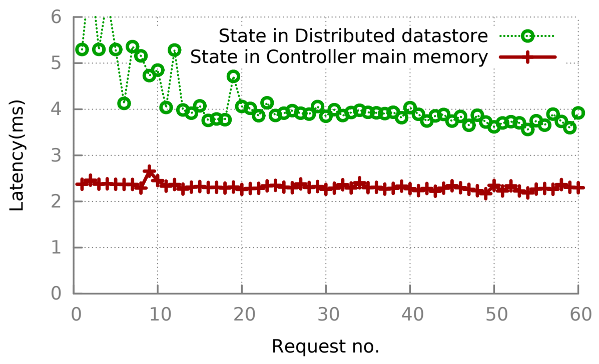


Figure 1: RTT for first packet of a ping request when tunnel data is accessed from a distributed data store and from in-memory

from the applications we surveyed, computation time spent during flow setup was negligible and communication time to access state from a distributed data store or to install flow rules was the key factor in flow setup latency.

In summary, we argue that, unlike existing distributed controller architectures, application state should be stored at controller instances, *and* SDN switches and application state should be dynamically assigned to controller instances. Both are critical to simultaneously minimizing flow setup latency and minimizing controller operating costs.

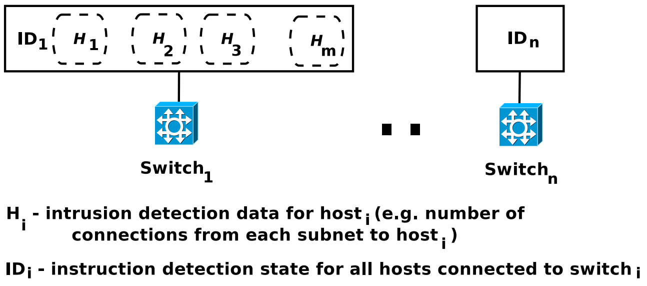
Localizing Event Handling and State. Schmid *et al.* [12] articulated the implications of using local algorithms to develop efficient coordination protocols in which each controller instance only needs to respond to events that take place in its local neighborhood. They solve an orthogonal problem by using near-optimal local algorithms for inherently global tasks to minimize the running time of the algorithm or to improve the performance. Our work focuses on partitioning applications' state and assigning switches to controllers to exploit such locality. The partitioning and aggregation logic of Onix [9] suggests to partition only when the size of application state (Network Information Base—tunnel data in this example) exceeds the capacity of a single Onix instance (coarse granularity) and would suffer from the static partitioning problem.

3. PRATYAASTHA ARCHITECTURE

We begin this section by illustrating our approach to partitioning application state by capturing dependencies with switches (Section 3.1). Then, we model and develop an algorithm to solve the problem of efficiently assigning state partitions and switches to controller instances such that (1) inter-controller communication is minimized, thereby keeping flow setup latencies low, and (2) the number of machines used to run controller instances is minimized, thereby keeping operating costs low (Section 3.2). Thereafter, we describe the elastic scaling workflow in our system (Section 3.4). Finally, we discuss how to leverage this architecture when running multiple applications in parallel (Section 3.5).

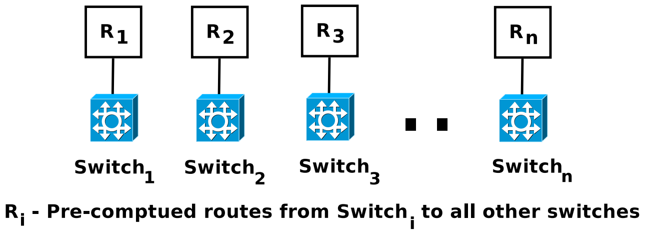
3.1 Partitioning Application State

Prior to assigning application state and SDN switches to controller instances, we must determine the granularity at which application state should be partitioned. For a given application, there may be several reasonable ways to divide its state: e.g., a multi-tenant virtualized data center application maintains state for individual tunnels, each of which is associated with a specific tenant;



ID_1
H_1
H_2
H_3
H_m
ID_n
$Switch_1$
$Switch_n$
H_i - intrusion detection data for host $_i$ (e.g. number of connections from each subnet to host $_i$)
ID_i - instruction detection state for all hosts connected to switch $_i$

Figure 2: State partitioning for an intrusion detection application



R_1
R_2
R_3
R_n
$Switch_1$
$Switch_2$
$Switch_3$
$Switch_n$
R_i - Pre-comptued routes from $Switch_i$ to all other switches

Figure 3: State partitioning for a traffic engineering application

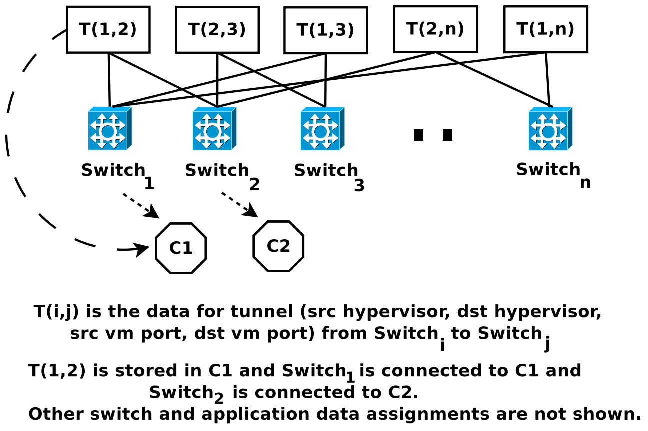
we could partition this state at the granularity of groups of tenants, individual tenants, groups of tunnels, or individual tunnels.

Choosing a suitable partitioning granularity is critical to finding an assignment that optimally satisfies the dual objectives discussed above. A partitioning granularity that is too coarse reduces the set of potential assignments, thereby limiting the extent to which we can find an assignment that satisfies our goals. In contrast, too fine of a partitioning granularity unnecessarily increases the complexity of solving the assignment problem, thereby limiting our ability to find an optimal assignment in bounded time.

An ideal partitioning of application state *minimizes the number of dependencies between SDN switches and a specific state partition*. We say a switch *depends* on a state partition if handling the events from the switch or invoking control actions on that switch requires accessing/updating application state contained in the partition. Applications can tag state objects with identifiers for pertinent switches so we can determine such dependencies; in the future, we plan to explore how we can leverage program analysis techniques to automatically infer such dependencies from an application's code.

Given this definition, we can define the partitioning problem in terms of a bipartite graph. A set of switches and a set of state partitions (determined based on a particular partitioning strategy) form the vertices. Edges between switches and partitions indicate a dependency, and the weight of the edge represents the amount of state that must be transferred between controller instances if the switch is assigned to a different instance than the state partition. The optimality of the partitioning is determined by the degree of the state partition vertices; ideally, we want all partition vertices to have degree one. If a given state partition has a degree greater than one *and* we could reasonably divide the partition into smaller pieces (e.g., the partition is currently a set of objects which could be broken into smaller sets), then the partition should be further divided and the bipartite graph updated.

There are ample applications where this ideal partitioning can be found. For example, an intrusion detection application that detects DoS attacks and port scans for hosts under the switches it controls



T(1,2)
T(2,3)
T(1,3)
T(2,n)
T(1,n)
$Switch_1$
$Switch_2$
$Switch_3$
$Switch_n$
C1
C2
T(i,j) is the data for tunnel (src hypervisor, dst hypervisor, src vm port, dst vm port) from $Switch_i$ to $Switch_j$
T(1,2) is stored in C1 and $Switch_1$ is connected to C1 and $Switch_2$ is connected to C2.
Other switch and application data assignments are not shown.

Figure 4: Application state for multi-tenant virtualized data center application

can group hosts, and the state associated with them, based on the switch to which the hosts are connected (Figure 2). We could instead partition state at the granularity of individual hosts, but this complicates the assignment problem and offers little benefit. As another example, the state for a traffic engineering application [14], which pre-computes routes and path weights between every source-destination pair, can be partitioned on the basis of the source switch (Figure 3).

However, there are applications—e.g., multi-tenant virtualized data center, middlebox orchestrator for clouds [7], and load balancing [15]—for which an ideal partitioning is not possible. This is due to the fact that some state for these applications is inherently associated with multiple switches: e.g., state for a tunnel between two switches may be accessed or updated when processing events from either switch (Figure 4).

Pooling. To reduce the number of dependencies between switches and state partitions, we can organize network switches and logically entities (e.g., tenants) into pools. For example, in a multi-tenant virtualized data center, the data center can be organized such that tenants are grouped into pools (say 20 tenants form a pool; the actual number may be decided by operators), and VMs of tenants are co-located only with VMs of other tenants belonging to the same pool. In other words, only tenants from the same pool share a hypervisor. Thus, the virtual switches in a pool's hypervisors will only depend on a single state partition that encompasses all application state for tenants in the pool. Such pooling and partitioning will give favorable options to the assignment algorithm to drastically minimize the inter-controller communication, assuming the pool's state partition and switches are placed on the same controller. Additionally, choosing smaller pool sizes will help in better packing efficiency in controllers. We acknowledge the fact that there is a trade-off between achieving better performance (by varying pool sizes) and the ability to place VMs anywhere, but this trade-off could be worthwhile.

3.2 Controller assignment problem

After partitioning, we need to figure out a way to optimally assign switches and application state partitions to controller instances. We want the assignment to minimize both the number of machines used and the amount of inter-controller communication. The former helps reduce costs for operating the controller instances, and the latter helps meet the latency requirements of SDN applications.

The controller assignment problem is a variant of the multi-dimensional bin packing problem. We model this problem as an integer linear program (Section 3.2.1) and we provide a heuristic approach to solve it (Section 3.2.2).

3.2.1 ILP formulation

We formulate the controller assignment problem as an integer linear program (ILP). The objective is to find an assignment that minimizes the weighted sum of the cost of controller machines utilized and the cost of inter-controller communications.

Inputs. Let P be the set of state partitions, and p_i be the memory requirement of the partition i. Let S be the set of switches, and s_i be the compute requirement of switch i. Let d_{ij} be the volume of state that must be transferred between controllers if switch i and state partition j are assigned to different controllers. Finally, let B be the set of machines (or bins) available for running controllers, w_i be the cost of machine i, and c_i and m_i be the compute and memory resources available at machine i.

Variables. The binary variables x_{ik} and y_{jk} are used to capture the controller assignment. They indicate whether switch i is mapped to resource k and whether partition j is mapped to resource k respectively. The binary variable z_i indicates whether the resource i is used as a controller instance, i.e., whether at least one switch or partition is assigned to resource i. The binary variable t_{ij} indicates whether switch i and partition j are assigned to different resources.

ILP Formulation.

minimize

$$\alpha \sum_{i \in B} z_i w_i + \beta \sum_{\substack{i \in S \\ j \in P}} d_{ij} t_{ij} \tag{1}$$

subject to

$$\sum_{i \in S} x_{ij} s_i \le c_j \qquad \forall j \in B \tag{2}$$

$$\sum_{i \in P} y_{ij} p_i \le m_j \qquad \forall j \in B \tag{3}$$

$$\sum_{j \in B} x_{ij} = 1 \qquad \forall i \in S \tag{4}$$

$$\sum_{j \in B} y_{ij} = 1 \qquad \forall i \in P \tag{5}$$

$$\sum_{i \in S} x_{ik} + \sum_{j \in P} y_{jk} \le M z_k \qquad \forall k \in B \tag{6}$$

$$t_{ij} \ge 1 - \sum_{k \in B} t'_{ijk} \qquad \forall i \in S, j \in P \tag{7}$$

$$t'_{ijk} \le x_{ik} \qquad \forall i \in S, j \in P, k \in B \tag{8}$$

$$t'_{ijk} \le y_{jk} \qquad \forall i \in S, j \in P, k \in B \tag{9}$$

$$x_{ij} \in \{0,1\} \qquad \forall i \in S, j \in B \tag{10}$$

$$y_{ij} \in \{0,1\} \qquad \forall i \in P, j \in B \tag{11}$$

$$t_{ij} \in \{0,1\} \qquad \forall i \in S, j \in P \tag{12}$$

$$t'_{ijk} \in \{0,1\} \qquad \forall i \in S, j \in P, k \in B \tag{13}$$

$$z_i \in \{0,1\} \qquad \forall i \in B \tag{14}$$

Objective. The first term ($\sum_{i \in B} z_i w_i$) in the objective function (eq. 1) captures the cost of resources used; the second term ($\sum_{\substack{i \in S \\ j \in P}} d_{ij} t_{ij}$) captures the communication cost between controllers when dependent switches and partitions are assigned to different resources. The weights α and β in the objective function help the network operator specify the relative importance of reducing resource costs and reducing inter-controller communication.

Constraints. Eq. 2 and eq. 3 specify the compute and memory limits of resources. Eq. 4 and eq. 5 makes sure all switches and partitions are assigned exactly one controller resource. Eq. 6, with $M \ge |S| + |P|$, helps in setting the variable z_k to 1 when at least one switch or partition is assigned to resource k. Eq. 7 to eq. 9 help in setting variable t_{ij} to 1 when switch i and partition j are placed on different controller resources; t'_{ijk} will be set to 1 when both switch i and partition j are assigned to the same controller resource k. Lastly, eq. 10 to eq. 14 indicates x_{ij}, y_{ij}, t_{ij}, t'_{ijk} and z_k are binary variables.

3.2.2 Heuristic approach

Accurately solving the ILP for a large topology (e.g., $O(10^4)$ switches, $O(10^2)$ controller machines) is not scalable. Hence, we model the ILP as a search problem. We use a local search algorithm (hill climbing with simulated annealing) to find the optimal assignment within a given time limit (e.g., 30 seconds). The time limit allows us to trade-off efficiency vs. reactivity: more search time allows us to find a more optimal assignment, leading to better efficiency, but high load in the meantime can impact flow setup latencies. The assignment found using a first-fit decreasing heuristic is set as the initial state during the hill climbing search algorithm.

3.3 Controller reassignment problem

The techniques in the previous sub-section help in assigning state partitions and switches to controller resources. However, during the operation of the network, the resource requirements of switches and state partitions can dynamically change. We need to adapt to such dynamic changes in resource requirements by reassigning some switches and partitions to existing or new controller resources. However, reassigning all switches and partitions is infeasible, because migrating state partitions, and reconfiguring switches to talk to the new controller instance, are expensive processes. Thus, the goal of reassignment should be to minimize the number of reassignments while at the same time utilizing minimal resources and minimizing inter-controller communication.

We can model this reassignment problem as an integer linear program (ILP), where the program takes as input the cost of reassignment of switches and state partitions, and minimizes the weighted sum of the cost of controller resources, the cost of inter-controller communications, and the cost of reassignment of switches and partitions. We can use a heuristic approach similar to the one in Section 3.2.2 to compute the reassignments efficiently. We plan to model and solve this problem as part of our future work.

3.4 Elastic Scaling

Initially, the application state is partitioned at the finest possible granularity (Section 3.1) and state partitions and switches are assigned to controller instances as per the assignment algorithm (Section 3.2). The controller instances are launched in the machines selected by the assignment algorithm. Each controller instance reports the flow-arrival rate from switches and the memory usage of state partitions it hosts. When one or more controllers are overloaded, we invoke the controller reassignment algorithm (Section 3.3) to find a new controller assignment for state partitions and switches. As per the new assignments, new controller instances are launched (in new machines) or unneeded controller instances are removed. The application state partitions are also migrated between controller machines. The switch migration protocol in ElastiCon [6] can be leveraged to migrate switches; migration

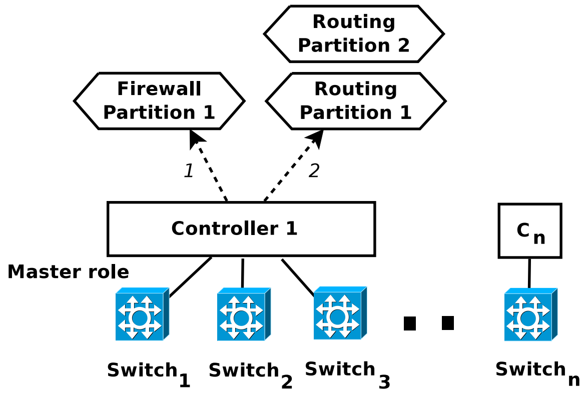


Figure 5: Assignment of multiple applications with co-ordination

only takes ≈20ms, during which flow setup latency increases by <1ms, so performance is not impacted much.

3.5 Multiple applications

There can be multiple SDN applications deployed in the same control plane (traffic engineering, multi-tenant virtualized data center, firewall, traffic accounting, etc.) and each application might have a different dependency graph (Section 3.1). We propose a generalized solution for the following two situations: (i) applications require coordination—e.g., a new flow needs to be processed by application 1 (firewall) followed by application 2 (routing); and (ii) applications operate independently without any conflicts— e.g., application 1 (routing) and an off-path application 2 (traffic accounting).

For (i), we run the controller assignment algorithm considering the memory requirements of all applications together. The orchestration layer in the controller sends network events (e.g., packet-in) to applications in the order specified by network policies. For instance, as shown in Figure 5, the controller sends a packet-in event to the firewall application (1) and, if the packet should not be blocked, subsequently sends the event to the routing application (2). The application partitions for firewall and routing might reside in different controller instances, so there may be RPC calls involved in such scenarios.

For (ii), we run the controller assignment algorithm separately for each application and dedicate individual controller instances for each application. A switch is connected to one controller (hosting partitions of one application) in a master role and connected to other controllers (each controller hosts application partitions of other applications) in an equal role. The packet-in events are sent in parallel to master and equal controllers and the partitions of different applications residing in controllers can act on the events *in parallel* without any coordination. This scenario can be seen in the example in Figure 6, where packet-in events from $Switch_1$ are sent *in parallel* to Controller 1 hosting a routing application (1) and to Controller 1' hosting a traffic accounting application (1').

4. EVALUATION

We evaluate Pratyaastha from two perspectives: (1) minimizing inter-controller communication to limit the number of flows with high setup latency; and (2) minimizing controller operating costs through efficient resource allocation.

We built a simulator written in Java to compare the performance and efficiency of Pratyaastha with other controller-assignment approaches. We implemented the controller assignment algorithm using the Optaplanner [3] planning engine. We used the 34 ToR topology and traffic characteristics of a private data center from Benson *et al.* [4]. For controller machines, we used the resource configurations (m1.small, m1.xlarge, m3.medium, m3.large, m3.xlarge, m3.2xlarge) and costs from Amazon EC2 instances [1].

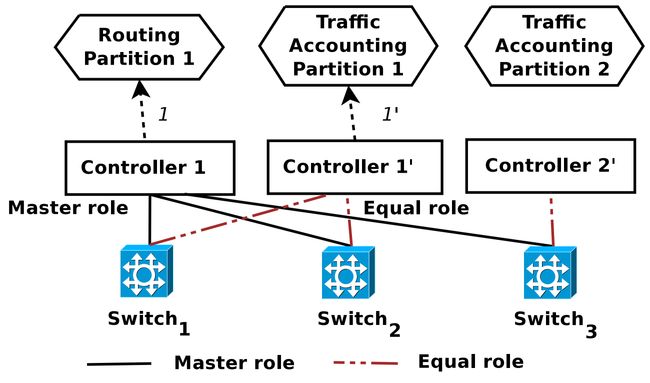


Figure 6: Assignment of multiple independent applications with no co-ordination

We initially partitioned the network topology into 11 pools, with 3 switches per pool (except the last pool, which has 4 switches); this is representative of the pooling that we advocate applying in the context of a multi-tenant virtualized data center application (Section 3.1). We randomly chose how much static data is contained in the state partition associated with each pool; this represents the memory consumed by application state like tunnel data and pre-computed routes, which do not change with the number of flows. Each switch generates traffic according to the flow inter-arrival time CDF reported in Benson *et al.* [4]. The dynamic memory used by a state partition is set in proportion to the flow-arrival rate at the switches in the associated pool; this represents the memory consumed by applications which store flow-level data, e.g., an intrusion detection application.

4.1 Performance

We used a value of 1 for α and β in the objective function in the assignment algorithm (Section 3.2). Pratyaastha's assignment algorithm (Section 3.2) and pooling technique (Section 3.1) helped in assigning almost 80% of related switches and state partitions to the same controller instance. This avoids costly RPC calls between controllers when handling events from 80% of the switches, thereby ensuring suitable flow setup latency for the flows at these switches. Furthermore, the assignment algorithm ensures that the processing responsibilities of a controller instance is within its CPU capacity, thereby avoiding overload. We note that by varying values for α and β, we can make efficiency vs. performance trade-offs. We omit details for brevity.

4.2 Resource utilization

We compare the resource utilization of Pratyaastha against two other designs: (1) *CPU only*, a system which only considers the flow-arrival rate at switches when determining a controller assignment, and which stores all state in a distributed data store; (2) *Local CPU + Mem*, a system which considers both the flow-arrival rate at switches and the dynamic memory usage of applications (for flow-level data), and which stores only static application data (e.g., tenant tunnel data) in a distributed data store. The CDFs of resource utilization of the three controller-assignment approaches are shown in Figure 7. Overall with Pratyaastha we get a 33% and a 42% decrease in cost when compared with *Local CPU + Mem* and *CPU only*, respectively. In the experiments, we allocated dedicated ma-

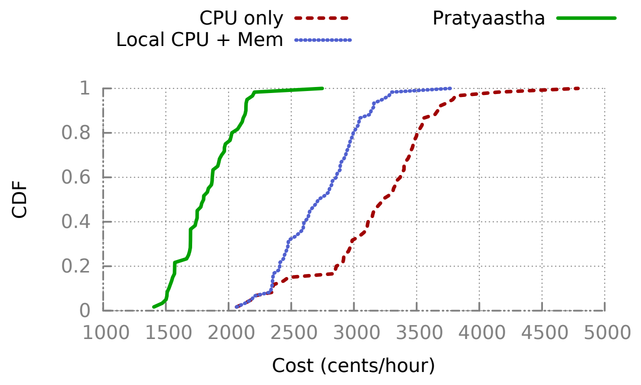


Figure 7: CDFs of resource utilization of CPU only, Local CPU + Mem, and Pratyaastha for a private data center topology with simulated flow arrival rates.

chines for the distributed data store, which are separate from the machines for controller instances. We could spread the distributed data store among the controller instances for (1) and (2), but both still suffer from latency overhead because of inter-controller communications and remote accesses to the distributed data store.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have shown that current state of the art distributed controller architectures are not sufficient to provide low flow setup latencies, and they incur more than optimal operating costs. We presented *Pratyaastha,* an efficient and elastic distributed SDN control plane that jointly minimizes inter-controller communication and resource consumption to address the above issues while allowing operators to make performance vs. cost tradeoffs. Our initial evaluation results are very promising, and we plan to build a full fledged system with fault tolerance and controller reassignment capabilities.

6. ACKNOWLEDGMENTS

We thank Aditya Akella, Junaid Khalid, Paul Suganthan G.C., and the anonymous reviewers for their valuable feedback. This work is supported in part by a Wisconsin Alumni Research Foundation (WARF) Accelerator Award and National Science Foundation grants CNS-1302041, CNS-1314363 and CNS-1040757. Aaron Gember-Jacobson is supported by an IBM PhD Fellowship.

7. REFERENCES

[1] EC2 pricing. `http://aws.amazon.com/ec2/pricing`.

[2] HP Network Protector SDN Application. `http://h17007.www1.hp.com/us/en/networking/products/network-management/Network_Protector_SDN_Application_Series`.

[3] OptaPlanner. `http://optaplanner.org`.

[4] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, 2010.

[5] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *SIGCOMM*, 2007.

[6] Advait Abhay Dixit, Fang Hao, Sarit Mukherjee, T. V. Lakshman, and Ramana Rao Kompella. Towards an elastic distributed SDN controller. In *HotSDN*, 2013.

[7] Aaron Gember, Anand Krishnamurthy, Saul St. John, Robert Grandl, Xiaoyang Gao, Ashok Anand, Theophilus Benson, Aditya Akella, and Vyas Sekar. Stratos: A network-aware orchestration layer for middleboxes in the cloud. Technical Report arXiv:1305.0209, 2013.

[8] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: A framework for efficient and scalable offloading of control applications. In *HotSDN*, 2012.

[9] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A distributed control platform for large-scale production networks. In *OSDI*, 2010.

[10] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[11] Matthew Monaco, Oliver Michel, and Eric Keller. Applying operating system principles to SDN controller design. In *HotNets*, 2013.

[12] Stefan Schmid and Jukka Suomela. Exploiting locality in distributed sdn control. In *HotSDN*, 2013.

[13] Amin Tootoonchian and Yashar Ganjali. HyperFlow: A distributed control plane for OpenFlow. In *INM/WREN*, 2010.

[14] Ramona Trestian, Gabriel-Miro Muntean, and Kostas Katrinis. MiceTrap: Scalable traffic engineering of datacenter mice flows using OpenFlow. In *IFIP/IEEE IM*, 2013.

[15] Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-based server load balancing gone wild. In *Hot-ICE*, 2011.