HAVEN: Holistic Load Balancing and Auto Scaling in the Cloud

Rishabh Poddar
IBM Research
rishabh.poddar@in.ibm.com

Anilkumar Vishnoi
IBM Research
vishnoianil@gmail.com

Vijay Mann
IBM Research
vijay.mann@in.ibm.com

***Abstract*—Load balancing and auto scaling are important services in the cloud. Traditionally, load balancing is achieved through either hardware or software appliances. Hardware appliances perform well but have several drawbacks. They are fairly expensive and are typically bought for managing peaks even if average volumes are 10% of peak. Further, they lack flexibility in terms of adding custom load balancing algorithms. They also lack multi-tenancy support. To address these concerns, most public clouds have adopted software load balancers that typically also comprise an auto scaling service. However, software load balancers do not match the performance of hardware load balancers. In order to avoid a single point of failure, they also require complex clustering solutions which further drives their cost higher. In this context, we present HAVEN—a system for holistic load balancing and auto scaling in a multi-tenant cloud environment that is naturally distributed, and hence scalable. It supports multi-tenancy and takes into account the utilization levels of different resources as part of its load balancing and auto scaling algorithms. HAVEN leverages software-defined networking to ensure that while the load balancing algorithm (control plane) executes on a server running network controller software, the packets to be load balanced never leave the data plane. For this reason, HAVEN is able to provide performance at par with a hardware load balancer while still providing the flexibility and customizability of a software load balancer. We validate HAVEN on a hardware setup and our experiments confirm that it achieves high performance without any significant overheads.**

I. INTRODUCTION

Load balancing and auto scaling are fundamental services in the cloud, around which other services are offered. They are critical to any highly scalable cloud deployment. Amazon realized it long ago and launched these services in its EC2 public cloud [1] in 2009. Google and Microsoft caught up much later and added these services to their cloud offerings in 2013 [2]. Many companies that run their services on these public cloud offerings have highlighted how auto scaling and load balancing have been critical to their core operations [3].

Given the extremely crucial role load balancing and auto scaling play in the cloud, they deserve to be investigated critically by both the systems and networking research communities. Traditionally, load balancing is achieved either through hardware or software appliances. Hardware appliances [4], [5] perform well but have several drawbacks. They are fairly expensive and are typically bought for managing peaks even if average volumes are 10% of peak. Typical, state-of-the-art hardware load balancers cost roughly US $80,000 for 20Gbps capacity [6]. Since most traffic must pass through the load balancer, a load balancer has to be provisioned for 100s of Gbps if not more. For example, authors in [6] show that for a 40,000 server network, built using a 2 level Clos network architecture, 400 Gbps of external traffic and 100 Tbps of intra-DC traffic will need load balancing or NAT. This will easily require 40 load balancers just for the external traffic (costing US $3.2 million) and 10,000 load balancers for the intra-DC traffic (costing US $800 million). To put these numbers in perspective, the total cost of load balancers (US $803.2 million) alone is enough to buy roughly 320,000 servers (assuming a typical list price of US $2500 per server [6]). Further, hardware load balancers lack flexibility in terms of adding custom load balancing algorithms and typically comprise 3-4 standard algorithms such as round robin, weighted round robin, least connections and least response time [7]. Typical hardware load balancers also lack multi-tenancy support.

978-1-4244-8953-4/11/$26.00

To address the above shortcomings of hardware load balancers, most public clouds have adopted software load balancers. These software load balancers comprise load balancing software running on a general purpose server or a virtual machine [8], [9]. They typically also consist of an auto scaling service, since all new instances that are added or deleted as a result of auto scaling have to be added or removed from the load balancing loop. While there are no reference performance numbers available, pure software load balancers present a huge challenge in terms of extracting performance comparable to hardware load balancers, because of limitations associated with packet processing at the user level in software, and the port capacity and density available on general purpose servers. In order to meet this performance challenge as well as to avoid a single point of failure, software load balancers require complex clustering solutions which further drives their cost higher.

In this paper, we present HAVEN—a system for holistic load balancing and auto scaling in a multi-tenant cloud environment that is naturally distributed and hence scalable. Unlike hardware and software load balancers, HAVEN does not involve an extra hop or a middlebox through which all traffic needs to pass. It supports multi-tenancy and takes into account the utilization levels of different resources in the cloud as part of its load balancing and auto scaling algorithms. HAVEN leverages software-defined networking to ensure that while the load balancing algorithm (control plane) executes on a server running network controller software, the packets to be load balanced never leave the data plane. For this reason, HAVEN is able to provide performance at par with a hardware load balancer while still providing the flexibility and customizability of a software load balancer. We present the overall design

of Haven in (§ III) and describe our architecture using an OpenDaylight [10] based controller in an OpenStack [11] environment in (§ IV). We validate HAVEN on a hardware setup in (§ V) and our experiments confirm that it achieves high performance without any significant overheads.

II. BACKGROUND

This section provides a technical background to the terms and technologies that form the basis of this paper.

A. Load Balancing and Auto Scaling

Load balancing is a key service in the cloud and refers to the routing of packets from a source to a chosen destination. A load balancer in the cloud will typically operate at layer 4 (TCP) or layer 7 (Application/HTTP). In this paper, we restrict ourselves to layer 4 load balancing. A layer 4 load balancer will spread incoming TCP connection requests over a load balanced pool of replica servers, such that all packets belonging to a given connection are always routed to the same chosen replica. A load balanced pool of servers will typically have a virtual IP (VIP) for the entire group, and an actual IP (AIP) per server. For a given connection request to a VIP (identified by the service's TCP port, and the client's IP address and TCP port), the load balancer translates the VIP into the AIP of the chosen server. The policy used to select the server is referred to as the load balancing policy.

Auto scaling helps in scaling a system horizontally in presence of a load spike by adding more instances of the application that can serve incoming requests. When load reduces or goes back to its normal state, some of the running instances of the application can then be stopped to ensure that the cloud tenant does not incur extra costs for running idle instances (since most public clouds charge by the time an instance runs). Auto scaling is best implemented as part of the load balancer service itself. This is because anytime new replicas are added to or removed from a load balanced pool, the load balancer needs to be notified so that it can add them to or remove them from its load balancing loop.

B. Software-Defined Networking and OpenFlow

Software-defined networking (SDN) is an emerging paradigm that consists of a network control plane that is physically decoupled from the data plane. The control plane runs as software on a commodity server, while the data plane is implemented in the switches. The control plane acts as the brain of the network and runs forwarding logic and policies, whereas the data plane acts on instructions from the control plane to forward the packets in the network. The control plane communicates with the data plane on its south-bound interface using a standard protocol such as OpenFlow. The control plane also exposes a north-bound API for 3rd party applications. Figure 1 shows the schematic architecture of an SDN setup.

OpenFlow is an open standard that is used by a network controller (control plane) to communicate with the data plane. OpenFlow rules installed by the controller dictate the forwarding logic of a switch. These rules typically have (i) a match portion that specifies which packets would match that rule, and (ii) an action portion that specifies what the data plane should do with matched packets. Actions can be either drop, forward

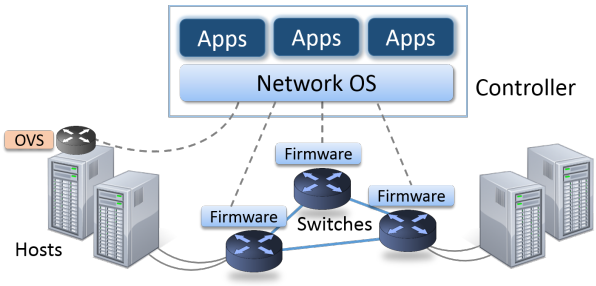

Fig. 1: SDN architecture

to one or more ports, or rewrite certain fields in the packet header. If an incoming packet does not match any existing rule at a switch, the default action for the switch is to forward the packet to the controller (referred to as a `PACKET_IN` event). The controller can then analyze the packet and install a matching rule for handling subsequent packets of that flow. To implement load balancing, an OpenFlow rule can thus be used to replace the VIP in an incoming request with the AIP of a server chosen from the load-balanced pool.

III. DESIGN

Since the task of load balancing is essentially directed towards effective and optimal usage of all available resources in the cloud, it is desirable for any holistic load balancing scheme to have the following properties:

- **Distributed:** A distributed system not only ensures that there is no single point of failure, but also avoids bottlenecks due to middleboxes and constrained routing. If the switches themselves can do load balancing, then the system also becomes naturally scalable.
- **Flexible:** If the load balancing technique is logically centralized, then it becomes easy for each service to avail of the technique with configurations specific to its own requirements.
- **Dynamic:** The load balancer should respond to changes in resource utilization patterns of the servers as well as the network, and the selection of a replica to load balance traffic must be informed by these changes.
- **Adaptive:** An effective auto scaling mechanism ensures that the system adapts to changes in demand, i.e., maintains performance during demand spikes, and reduces the tenant's cost during periods of inactivity. Further, the selection of a suitable replica while scaling up or down is also important.

The separation of control and data planes in SDNs allows switches to perform the load balancing, while the network behavior remains logically centralized at the controller. This not only makes the system naturally scalable, but also highly flexible as the load balancing technique can be defined centrally in the control plane for the entire network. HAVEN has been designed as an OpenFlow controller application, and encapsulates the principles listed above. In essence, HAVEN monitors the congestion levels in the network links and orchestrates the collection of resource usage statistics from the cloud, such as the CPU and memory utilization levels of the provisioned virtual machines (i.e., the servers). It uses these measurements to periodically compute scores for all the virtual

```
Input: 𝕍: set of all pool members, 𝕃: set of network links
function StatisticsManager(𝕍, 𝕃)
    /*Collect statistics*/
    CollectBwStats(𝕃)
    CollectCpuUtil(𝕍)
    CollectMemUtil(𝕍)
    /*Notify listeners of update()*/
    Notify()
end function
```

Fig. 2: Statistics Manager service

machines (VMs) in the cloud. When a request arrives for a VIP, HAVEN ranks all the member VMs in the tenant's pool on the basis of the computed scores and selects the one with the maximum available resources (network as well as server utilization). Simultaneously, HAVEN decides whether or not the pool should scale up or down based on the current demand.

A. Statistics Manager Service

HAVEN gathers two kinds of data from the cloud—link bandwidth statistics from the network, and resource utilization information from the provisioned VMs (Figure 2). The link bandwidth data are computed from the OpenFlow `STATS_REPLY` messages that are periodically collected from the switches in the network. These messages communicate statistics gathered at the switch per port, flow, and table (e.g., the total number of packets/bytes sent and received). Additionally, HAVEN also gathers resource usage patterns of all the VMs in the cloud—in particular, CPU and memory utilization levels—from the respective hypervisors.

B. Score Manager Service

Based on the measurements gathered by the Statistics Manager, HAVEN's Score Manager Service computes scores for all provisioned VMs whenever it receives an update notification from the Statistics Manager (Figure 3). This score is indicative of the resources available at the VM's disposal, which include (a) the available bandwidth on the paths to the VM from gateway switches, (b) % CPU utilization of the VM, and (c) % memory utilization of the VM. The lower the usage levels of a member's resources, the lower its score. Since there can be multiple gateway switches, VMs are accessible via multiple paths; hence, scores are assigned to a VM *per* path.

SCORE COMPUTATION. Figure 4 outlines HAVEN's score computation function. Given a resource with utilization value i, its relative importance towards the total score Σ_V is indicated by a user-defined weight w_i. The contribution of an over-utilized resource is magnified by squaring i to allow HAVEN to identify VMs with resource bottlenecks. Thus, a VM u with high CPU utilization β and low memory utilization γ will have a higher score than a VM v with moderate β and γ.

C. Load Balancer Service

When a `PACKET_IN` arrives from a client for a particular VIP, the load balancer component selects a VM from the tenant's pool based on the calculated scores. The service selects that VM which has the lowest score (and hence the one with the most available resources) and has been assigned to clients

```
Input: 𝕍: set of all pool members
Output: Σ: set of scores
function ScoreManager(𝕍)
    for all v ∈ 𝕍 do
        ℙ_v ← GetPaths(v)
        for all P ← ℙ_v do
            /*Compute scores*/
            α ← GetBwUtilization(P)
            β ← GetCpuUtilization(v)
            γ ← GetMemUtilization(v)
            Σ_{v,P} ← GetScore(α, β, γ)
            Σ ← Σ ∪ Σ_{v,P}
        end for
    end for
    /*Notify listeners of update()*/
    Notify()
    return Σ
end function
```

Fig. 3: Score Manager service

```
Input: α, β, γ: resource utilization values
Output: Σ_V: computed score
Constants: {w_i} = weights of parameters i ∈ {α, β, γ}
function GetScore(α, β, γ)
    Σ_V ← ∑(w_i · i²)
    return Σ_V
end function
```

Fig. 4: Score function

the least number of times since the last update of scores. Since score updates depend on the granularity of statistics collection (order of seconds), this ensures that requests are distributed fairly amongst the pool members between subsequent updates, and VMs are not choked during demand spikes. The service does this by maintaining a map of the number of flows assigned per VM between score updates, and flushes this map when it receives a score update notification. Figure 5 describes the methodology employed by HAVEN for load balancing.

D. Auto Scaling Service

On the arrival of a `PACKET_IN` for a VIP, HAVEN in parallel also decides whether or not the system needs to be scaled (up or down).

1) Scale-up Service: Figure 6 describes the scale-up decision methodology. The service takes as input the score Σ_V of the pool member selected for load balancing, and thus having the lowest overall resource utilization. If Σ_V is greater than a threshold value τ_{UP}, then the system is scaled up provided no other pool member is pending activation. Note that $\Sigma_V > \tau_{\text{UP}}$ would imply that the scores of all other pool members are also higher than τ_{UP}. The pool member whose host switch has the highest available bandwidth on its links is selected for activation, and is added to the tenant's list of pending members until it is completely launched.

2) Scale-down Service: Figure 7 describes the scale-down decision methodology. The service determines whether or not the system needs to be scaled down based on the resource utilization levels across all the active pool members. Further, only the last hop bandwidth utilization is considered for every

```
Input: 𝕍: set of all pool members, S_SRC: packet entry node, Θ: map
       of number of flows assigned per VM since last score update
Output: V: candidate member for load balancing
 1: function LoadBalancer(𝕍, S_SRC)
 2:     if ScoresUpdateReceived() = true then
 3:         /*Consider all pool members
 4:           and clear flow count map*/
 5:         𝕍_A ← GetAllActiveMembers(𝕍)
 6:         ClearFlowCounts(Θ)
 7:     else
 8:         /*Only consider members that have
 9:           been assigned least flows since
10:           last score update*/
11:         𝕍_A ← GetLeastFlowCountMembers(Θ, 𝕍)
12:     end if
13:     for all v ∈ 𝕍_A do
14:         P ← GetPath(S_SRC, v)
15:         /*Compute score for the member*/
16:         Σ_v ← GetScore(v, P)
17:     end for
18:     V ← v ∈ 𝕍_A such that Σ_v is minimum
19:     /*Update flow count map*/
20:     IncrementFlowCount(Θ, V)
21:     return V
22: end function
```

Fig. 5: HAVEN's load balancing algorithm

```
Input: 𝕍: set of all pool members, Σ_V: score of V, candidate member
       for load balancing
Output: V: candidate member for activation
Constants: τ_UP: threshold score for scale-up decision
 1: function ScaleUp(𝕍, Σ_V)
 2:     𝕍_P ← GetPendingMembers(𝕍)
 3:     𝕍_I ← GetInactiveMembers(𝕍)
 4:     if Σ_V ≥ τ_UP , |𝕍_P| = 0 and |𝕍_I| > 0 then
 5:         for all v ∈ 𝕍_I do
 6:             /*Get node b/w utilization*/
 7:             S_v ← GetHostNode(v)
 8:             𝕃 ← GetLinks(S_v)
 9:             α_v ← GetBwUtilization(𝕃)
10:         end for
11:         V ← v ∈ 𝕍_I such that α_v is minimum
12:     end if
13:     return V
14: end function
```

Fig. 6: Scale-up service

member while computing its score, so that traffic generated due to other entities in the network are not falsely ascribed to it. If majority of the active members have scores higher than a threshold value τ_{DOWN}, then the system is scaled down. The pool member with the longest uptime is selected for deactivation to provide for maintenance (such as patches/upgrades). Note that while the selection of a member for deactivation removes it from the tenant's list of active pool members, it does not imply its immediate shutdown. All active flows to the member must expire before it can be actually terminated, determined via the OpenFlow `FLOW_REMOVED` messages communicated to the controller by the network switches.

IMPLEMENTATION. Note that the algorithms described in Figures 2–7 needn't perform all computations in the critical path of execution. The collection of different statistics can

```
Input: 𝕍: set of all pool members
Output: V: candidate member for deactivation
Constants: τ_DOWN: threshold score for scale-down decision
 1: function ScaleDown(𝕍, V)
 2:     𝕍_A ← GetActiveMembers(𝕍)
 3:     c ← 0
 4:     for all v ∈ 𝕍_A do
 5:         /*Get last hop utilization*/
 6:         S_v ← GetHostNode(v)
 7:         l ← GetLink(S_v, v)
 8:         α ← GetBwUtilization(l)
 9:         /*Get VM resource utilization*/
10:         β ← GetCpuUtilization(v)
11:         γ ← GetMemUtilization(v)
12:         /*Compute score for the member*/
13:         Σ_v ← GetScore(α, β, γ)
14:         if Σ_v < τ_DOWN then
15:             c ← c + 1
16:         end if
17:     end for
18:     if c > |𝕍_A| /2 then
19:         V ← v ∈ 𝕍_A with longest uptime
20:     end if
21:     return V
22: end function
```

Fig. 7: Scale-down service

be offloaded to multiple threads executing in parallel, while scores are simultaneously computed and cached. Similarly, computation of routes can be offloaded to a background service that caches routes and updates them in the event of topological changes. Further, while load balancing and auto scaling are seemingly distinct activities, they can be conflated into a single routine. We employ these techniques in the actual implementation of HAVEN described in the next section.

IV. SYSTEM ARCHITECTURE

HAVEN has been implemented as an application for an OpenDaylight based controller integrated with OpenStack, an open-source cloud computing software platform as shown in Figure 8 shows our architecture. Implementation of HAVEN in an OpenStack environment has 3 major advantages. First, we install the address translation rule inside a vSwitch, allowing us to scale much better, since most physical switches do not allow more than 4K-6K OpenFlow rules. Second, we get access to OpenStack's telemetry capabilities enabling us to provide a holistic load balancer. Third, by leveraging the Nova VM provisioning service inside OpenStack, we are able to provide an integrated load balancing and auto scaling service. Currently, we use HAVEN primarily as an internal load balancer (where the client resides in the same data center or L2 domain as the destination) since we rely on the ingress vSwitch to do the address translation. This also prevents us from problems of having OpenFlow support in the border routers (for external load balancing, where the client is outside the data center) as well as other issues related to replacing BGP with centralized routing [6]. Nonetheless, since intra-DC traffic is a significant portion of the overall data center traffic, we believe HAVEN provides an extremely critical service in the cloud.

We next describe relevant components of the overall system along with HAVEN's implementation within the controller.

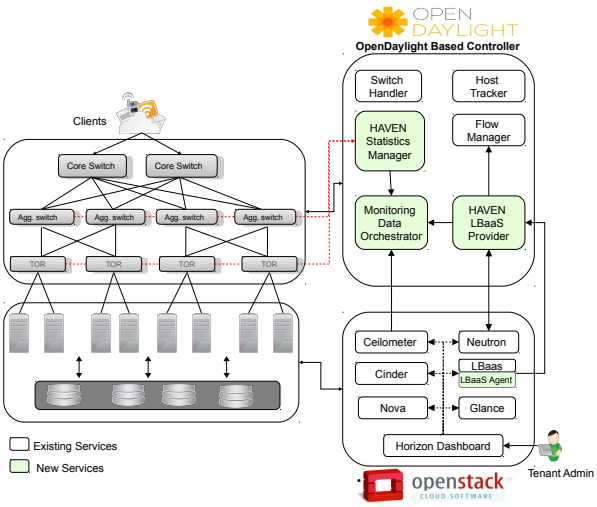

Fig. 8: HAVEN architecture.

A. OpenStack

OpenStack has a modular architecture with components for various services. We describe the components that are relevant to this paper.

1) Neutron: OpenStack's networking service is provided by Neutron, which provides tenants with APIs to build topologies and configure network policies in the cloud. Neutron provides for the usage of plugins (which are back-end implementations of the networking API) to allow different technologies to implement the logical API requests received by it (such as the creation of tenants, networks, and subnets). We use a plugin packaged with Neutron to communicate these requests to the network virtualization solution provided by controller.

2) LBaaS: The Load-Balancing-as-a-Service (LBaaS) plugin of Neutron allows the use of open-source load balancing technologies to drive the actual load balancing of requests. LBaaS requests (such as the allocation of pools, VIPs, and pool members) are received via REST APIs exposed by Neutron. We developed a service extension driver for the plugin to drive the communication of LBaaS data to the controller.

3) Ceilometer: Ceilometer is OpenStack's telemetry infrastructure that collects measurements from the cloud. Its primary functions are monitoring and metering of resources, and provides APIs for the retrieval of the collected data. HAVEN retrieves the VM usage statistics collected by Ceilometer via the REST APIs exposed by the service.

B. OpenDaylight based Controller

The controller contains a virtualization solution which communicates with the virtualization agent in Neutron to provide networking services. HAVEN in turn communicates with the virtualization solution to configure the network for the purposes of load balancing. The overall control flow given in Figure 9. We subsequently describe the components of HAVEN implemented as an application within the controller.

1) Monitoring Data Orchestrator: This module implements HAVEN's statistics manager (§ III-A) and score manager (§ III-B) services, and is responsible for the management of all statistics harvested from the cloud, both network as well as VM resources. The module periodically polls OpenStack's Ceilometer for CPU and memory utilization measurements gathered from all VMs that are part of some tenant's load balancing pool. Simultaneously, it collects link bandwidth statistics by periodically issuing `STATS_REQUEST` messages to the OpenFlow-enabled network switches. The collection of both kinds of data is performed at the granularity of five seconds. Once all the switches have responded with corresponding `STATS_REPLY` messages, scores are computed for all VMs *per* known path and are cached, and the LBaaS Provider is notified of the score update.

2) HAVEN LBaaS Provider: This is HAVEN's central module that handles the load balancing of packets for a particular VIP (§ III-C), and auto scaling (§ III-D) of a tenant's pool. When a request arrives for a tenant's VIP, the module in communication with the Data Orchestrator selects a VM for load balancing the traffic as described in Figure 5. It then installs flows in the network switches to load balance the flow to the VM with the best score. Simultaneously, the service also decides whether or not the tenant's pool needs to be scaled. The module also implements a northbound interface that exposes REST APIs invoked by Neutron's LBaaS agent for the communication of LBaaS data. The data is subsequently cached in HAVEN's local database and is used to configure the network for load balancing.

V. EVALUATION

In this section, we describe the evaluation of HAVEN as a holistic load balancer in an OpenStack environment. We evaluate the performance of our design by (i) comparing it with HAProxy, a popular open-source load balancer software [12], and (ii) assessing the overheads imposed by HAVEN on the network controller.

TESTBED. We deployed the Havana release of OpenStack on our physical testbed, consisting of 7 servers connected over a network of 14 OpenFlow-enabled switches (IBM RackSwitch G8264 [13]) arranged in a three-tiered design with 8 edge, 4 aggregate, and 2 core switches. All of our servers are IBM x3650 M3 machines having 2 Intel Xeon x5675 CPUs with 6 cores each (12 cores in total) at 3.07 GHz, and 128 GB of RAM, running 64 bit Ubuntu Linux v12.04. The cloud controller services (OpenStack Nova, Neutron, etc.) were run on a separate dedicated server of the same configuration, along with HAProxy/ODL controller.

TOOLS USED. We used httperf [14] to generate HTTP workloads of various sizes and drive traffic to web servers for load balancing. We used nmon, a system monitor tool for Linux [15], to track the CPU usage at the controller during our experiments. We also used the Ping utility to capture the overhead of using HAVEN on the latency of the first packets of flows that are sent to the controller as `PACKET_IN` messages.

A. Performance

HAVEN's architecture can be used to deploy different load balancing policies. We evaluated the performance of HAVEN by experimenting with two policies—(i) HAVEN–RA: a resource-aware policy (as discussed in § III), and

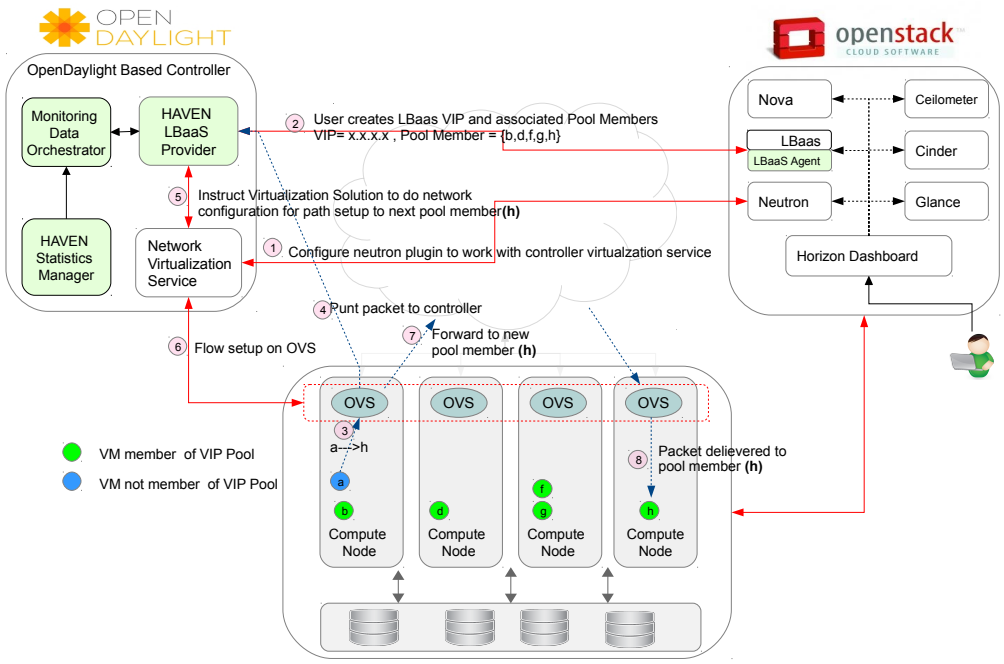

Fig. 9: HAVEN control flow.

(ii) HAVEN–RR: a round-robin policy implemented within HAVEN. For a large number of environments that do not have high variability in the workload, a round-robin policy might just about be sufficient. However, HAVEN's architecture ensures enhanced performance even in those cases by avoiding the extra hops that are typically involved for other software load balancers. Thus, we compare the performance of these two policies with that of an HAProxy-based approach, and present our findings.

EXPERIMENTAL SETUP. We deployed 4 VMs each on four out of our seven servers, and used the remaining three for generating client workloads. From each of these three client servers, we ran 5 parallel httperf sessions for files of 5 different sizes (one session per file type) located at the pool members—3 small files (1KB, 10KB, and 100KB) and two large files (1MB and 10MB). We used this setup to run four sets of experiments for each of the three policies—by varying the number of pool members between 8 and 16, and by varying the generated workload between high and low—described as follows:

(**1**) **16 members (high):** Pool size of 16 members; each httperf session configured for a total of 1800 connections, at the rate of 3 connections/sec, with 50 requests per connection for small files, and 20 requests per connection for large files, resulting in a request ratio of roughly 80:20 for small:large files.

(**2**) **16 members (low):** Pool size of 16 members; each httperf session configured for a total of 1200 connections, at the rate of 2 connections/sec, with 25 requests per connection for small files, and 10 requests per connection for large files.

(**3**) **8 members (high):** Pool size of 8 members; each httperf session configured for a total of 1800 connections, at the rate of 3 connections/sec, with 50 requests per connection for small files, and 20 requests per connection for large files.

(**4**) **8 members (low):** Pool size of 8 members; each httperf session configured for a total of 1200 connections, at the rate of 2 connections/sec, with 25 requests per connection for small files, and 10 requests per connection for large files.

RESULTS. Httperf output has 3 metrics that are of interest to us. These are (i) Average Reply Time - Response, which is the time spent between sending the last byte of the request and receiving the first byte of response, (ii) Average Reply Time - Transfer, which is the time spent between receiving the first byte of response and the last byte of response, and (iii) Average Reply Rate (per second), which is the rate at which the server responded. The results are shown in Figures 10, 11 and 12 respectively, as a weighted average of the measurements obtained for the httperf sessions corresponding to different file sizes. HAVEN outperfoms HAProxy in all metrics for both low and high traffic and for both pool sizes (16 and 8). By removing a middlebox (as in the case of HAProxy) and thereby eliminating bottleneck links and choke points, HAVEN is able to improve response times drastically by up to 50 times, while the transfer times are improved by a factor of 2.1. The reply rates or throughput are improved by a factor of 3.8.

B. Overheads

In order to assess the performance overheads of incurred by the deployment of HAVEN, we ran two sets of experiments on our testbed. In the first experiment, RTT measurements using the ping utility were used to assess the overhead of HAVEN on the latency of first packets of flows; in the second experiment, we plot the CPU usage at the controller while running a high workload on a pool of 16 members.

1) Ping packet latencies: We configured the controller to install flow rules with an idle timeout of 1 second. We then sent 200 ping packets from a client to a VIP, at an interval of 3 seconds. This ensured that each ping packet reached

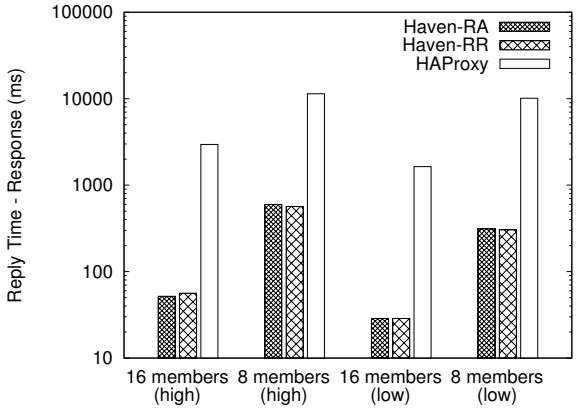

Fig. 10: Average Reply Time - Response: HAVEN outperforms HAProxy by up to 50 times

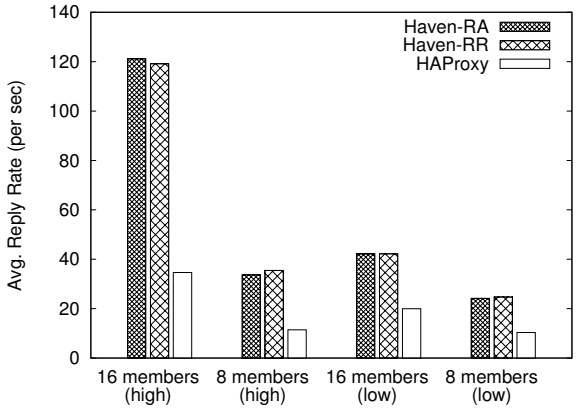

Fig. 12: Average Reply Rate (per second): HAVEN outperforms HAProxy by a factor of 3.8

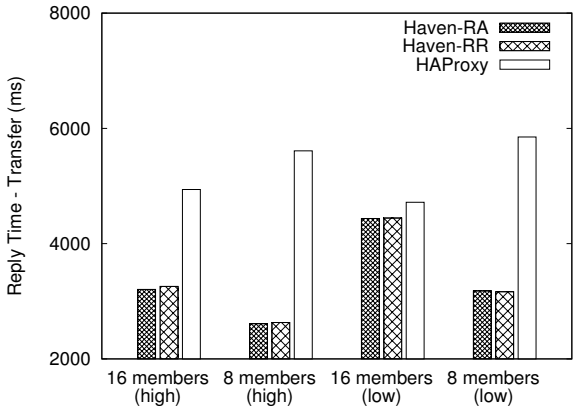

Fig. 11: Average Reply Time - Transfer: HAVEN outperforms HAProxy by a factor of 2.1

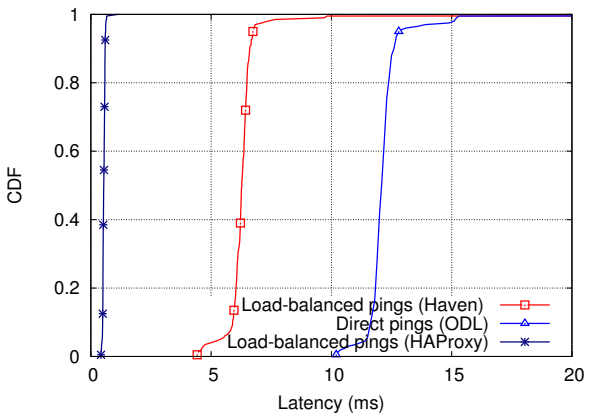

Fig. 13: Comparison of ping packet latencies

the controller as a PACKET_IN message, and was load-balanced by HAVEN. We measured the RTTs of the packets, and plotted their CDF. We repeated the experiment in 2 additional scenarios—(i) we sent ping packets directly at the IP address of a pool member (and not the pool's VIP); these packets were thus routed by the default forwarding module of the OpenDaylight (ODL) controller and not load balanced by HAVEN; (ii) we sent ping packets to a VIP using HAProxy (and not HAVEN). The results are presented in Figure 13.

RESULTS. In the presence of HAVEN, the first packet latency is approximately ~6ms. As expected, this is higher than the case with an HAProxy (~500μs), as in the latter no trips to the controller are involved. However, since this additional packet latency is limited to the first packets of flows, it has no bearing on the overall performance of HAVEN as compared with HAProxy, as seen in § V-A. Further, the latency incurred by packets load balanced by HAVEN are lesser than that the packets directly forwarded by ODL's default forwarding module. We attribute this to the fact that HAVEN computes routes offline and caches them, while the default forwarding module computes routes in the critical path of execution. Thus, HAVEN's modules do not incur substantial overheads on the performance of the controller.

C. Controller CPU usage

We further validate the fact that HAVEN has negligible overheads on the controller's performance by monitoring CPU usage for 700 seconds for two policies—HAVEN–RA and HAVEN–RR (as described in § V-A). We used a setup of 16 pool members with a high workload from 3 client servers. The results are presented in Figure 14, at intervals of 10 seconds. We observed that on an average, the CPU usage did not exceed ~8% utilization for HAVEN–RA policy, and ~7% for HAVEN–RR, well within permissible limits. Further, even during spikes of activity, it does not exceed 20% utilization. We attribute these spikes to JVM's garbage collection.

VI. RELATED WORK

Recent research systems [6], [16], [17] have achieved SDN based load balancing and auto scaling. Plug-n-Serve system [16] uses OpenFlow to reactively assign client requests to replicas based on the network and server load. This system is similar to HAVEN in that it intercepts the first packet of each client request and installs an individual forwarding rule that handles the remaining packets of that connection. However, we improve over Plug-n-Serve in several ways. We implement our solution in an OpenStack environment that allows us

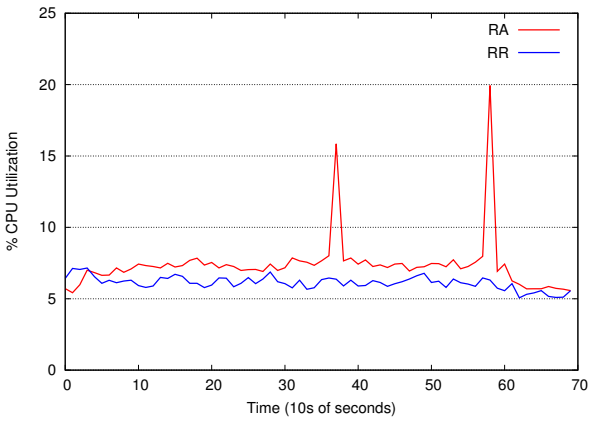

Fig. 14: CPU usage at controller

to make three significant improvements. First, we install the address translation rule inside a vSwitch that allows us to scale much better since most physical switches do not allow more than 4000-6000 OpenFlow rules. Second, we get access to OpenStack's telemetry capabilities that enables us to provide a holistic load balancer. Third, by leveraging the Nova VM provisioning service inside OpenStack, we are able to provide an integrated load balancing and auto scaling service.

Ananta [6] relies on SDN principles and leverages custom software running on commodity hardware servers. It has a distributed, scalable architecture for layer-4 load balancing and NAT. It is able to achieve a throughput of around 40 Tbps using 400 servers. It provides N+1 redundancy and quick failover along with tenant isolation. While Ananta is a great engineering effort, it is based on closed standards and it is unlikely that others in the community can easily replicate that effort. HAVEN tries to implement a holistic load balancing and auto scaling system using open standards such as OpenFlow and through standard network and cloud platforms such as OpenDaylight and OpenStack.

DUET [18] proposes a hybrid approach by integrating a switch-based load balancer design with software load balancing. It controls available functions in commodity switches (traffic splitting and packet encapsulation) to enable them to act as hardware load balancers, and uses a small deployment of software load balancers in case of switch failures. HAVEN, on the other hand, leverages an SDN-based approach to implement load balancing in the switches themselves while additionally accounting for resource utilization levels in the cloud, and also provides the added capability of auto scaling.

Wang et al. [17] propose a load balancing architecture using OpenFlow rules, that proactively maps blocks of source IP addresses to replica servers so that client requests are directly forwarded to them with minimal intervention by the controller. They propose a 'partitioning' algorithm that determines a minimal set of wildcard rules to install, and a 'transitioning' algorithm that changes these rules to adapt to new load balancing weights. We believe this work is complementary to our approach, and HAVEN can leverage these algorithms.

VII. CONCLUSION

In this paper, we present the design and implementation of HAVEN—a system that achieves holistic load balancing and auto scaling in the cloud. HAVEN leverages software defined networking to ensure that custom resource aware load balancing policies are added to the control plane, while most traffic never leaves the data plane. This helps in avoiding an extra hop or a middlebox that most other load balancing approaches involve. HAVEN has been implemented as a set of modules inside an OpenDaylight based OpenFlow controller that provides network service for an OpenStack managed cloud through an LBaaS driver extension that we developed. Our preliminary evaluation of HAVEN's load balancing capabilities on a hardware testbed demonstrates that HAVEN is faster than a standard software load balancer widely used in the industry. We are currently conducting a more comprehensive evaluation of HAVEN's load balancing and auto scaling capabilities in presence of higher workloads. We are also working on a production deployment of HAVEN in IBM data centers.

REFERENCES

[1] "New Features for Amazon EC2: Elastic Load Balancing, Auto Scaling, and Amazon CloudWatch," http://aws.amazon.com/blogs/aws/new-aws-load-balancing-automatic-scaling-and-cloud-monitoring-services/.

[2] "Google, Microsoft play catch up to Amazon, add load balancing, auto-scaling to their clouds," http://www.networkworld.com/article/2168844/cloud-computing/google--microsoft-play-catch-up-to-amazon--add-load-balancing--auto-scaling-to-their.html.

[3] "Auto Scaling in the Amazon Cloud," http://techblog.netflix.com/2012/01/auto-scaling-in-amazon-cloud.html.

[4] "F5 BIG-IP," http://www.f5.com.

[5] "A10 Networks AX Series," http://www.a10networks.com.

[6] P. Patel *et al.*, "Ananta: cloud scale load balancing," in *ACM SIGCOMM 2013*.

[7] "F5 Load Balancers," https://f5.com/glossary/load-balancer.

[8] "LoadBalancer.org Vitual Appliance," http://www.load-balancer.org.

[9] "NetScalar VPX Virtual Appliance," http://www.citrix.com.

[10] "OpenDaylight," http://www.opendaylight.org.

[11] "OpenStack," http://www.openstack.org/.

[12] "HA Proxy Load Balancer," http://haproxy.lwt.eu.

[13] "IBM System Networking RackSwitch G8264," http://www-03.ibm.com/systems/networking/switches/rack/g8264/.

[14] "Httperf," http://www.hpl.hp.com/research/linux/httperf/.

[15] "Nmon," http://nmon.sourceforge.net/.

[16] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari., "Plug-n-Serve: Load-balancing web traffic using OpenFlow," in *Demo at ACM SIGCOMM 2009*.

[17] R. Wang, D. Butnariu, and J. Rexford, "OpenFlow-based Server Load Balancing Gone Wild," in *Hot-ICE 2011*.

[18] R. Gandhi *et al.*, "DUET: Cloud Scale Load Balancing with Hardware and Software," in *ACM SIGCOMM 2014*.