A Scalable Multi-Datacenter Layer-2 Network Architecture

Chen Chen
University of Pennsylvania
chenche@cis.upenn.edu

Changbin Liu
AT&T Labs - Research
changbl@research.att.com

Pingkai Liu
AT&T Labs - Research
pingkai@research.att.com

Boon Thau Loo
University of Pennsylvania
boonloo@cis.upenn.edu

Ling Ding
University of Pennsylvania
lingding@cis.upenn.edu

ABSTRACT

Cloud today is evolving towards multi-datacenter deployment, with each datacenter serving customers in different geographical areas. The independence between datacenters, however, prohibits effective inter-datacenter resource sharing and flexible management of the infrastructure. In this paper, we propose WL2, a Software-Defined Networking (SDN) solution to an Internet-scale Layer-2 network across multiple datacenters. In WL2, a logically centralized controller handles control-plane communication and configuration in each datacenter. We achieve scalability in three ways: (1) eliminating Layer-2 broadcast by rerouting control-plane traffic to the controller; (2) introducing a layered addressing scheme for aggregate Layer-2 routing; and (3) creating an overlay abstraction on top of physical topology for fast flow setup. WL2 is fault-tolerant against controller and gateway failures. We deployed and evaluated WL2 in a 2,250-VM testbed across three datacenters. The results indicate high performance and robustness of the system.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Network communications*; C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network management*

Keywords

Software-defined networking, Layer-2 networking, multiple datacenters, scalability, fault-tolerance

1. INTRODUCTION

Cloud today is evolving towards multi-datacenter deployment over diverse geographical areas. For example, Google [20] and Amazon [1] both maintain global infrastructure of datacenters in different continents for better service quality (*e.g.,* low latency and high throughput) to their regional customers. Moreover, geographically distributed datacenters enable service continuity in face of large-scale disasters that could shut down a whole datacenter.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SOSR2015, June 17 - 18, 2015, Santa Clara, CA, USA
Copyright 2015 ACM . ISBN 978-1-4503-3451-8/15/06$15.00
DOI: http://dx.doi.org/10.1145/2774993.2775008.

However, current cloud infrastructure is more like *multiple clouds* rather than *a multi-datacenter cloud*, in that different datacenters usually operate independently except for data replication. This setup has several drawbacks. First, it is hard for network administrators to manage the infrastructure as a whole, prohibiting inter-datacenter optimizations such as load balancing and virtual machine (VM) migration. Second, customers, especially those that have business presence at different locations, may demand an inter-datacenter sub-network to achieve low-latency and seamless access from each location. This requirement cannot be easily satisfied by today's cloud.

A datacenter is often built as an IP network for scalability. However, this increases the complexity of network management by introducing substantial configuration overhead (*e.g.,* IP division, BGP routing). Moreover, since IP adopts the location-aware addressing scheme, it is difficult, if not possible, to perform VM live migration [17] across IP prefix boundaries. IP address change during VM migration would inevitably invalidate existing network sessions.

In contrast, Layer-2 (L2) network provides a simple abstraction for both users and administrators. Central to L2's ease of management is its plug-and-play semantics, which allows administrators to add or remove devices with minimal disruptions. Moreover, L2 network uses flat addressing scheme, which supports VM live migration in the network without disrupting ongoing network sessions [14, 16]. Public clouds including Amazon EC2 [13], Google Compute Engine [6], and Microsoft Azure [28] all present users with L2 network abstraction. OpenStack [31], the de facto standard of open cloud platforms, also adopts L2 network.

Despite its ease of configuration and management, L2 network does not scale, particularly under the scenario of wide-area networks (WANs). The major scalability bottleneck lies at the broadcast nature of control traffic in a L2 network, such as Spanning Tree Protocol (STP), Dynamic Host Configuration Protocol (DHCP) and Address Resolution Protocol (ARP). A number of unconventional L2 designs [18, 30, 34] have been proposed to address the scalability issues of L2 within a *single* datacenter. It remains challenging to design and implement a scalable L2 solution in the wide-area scenario, where multiple datacenters of potentially heterogeneous topologies are interconnected via the Internet.

In light of this, we present WL2 (*Wide-area Layer-2*), which provides a L2 network abstraction that spans multiple datacenters across diverse geographical regions, and yet maintains good scalability and high availability. Specifically, WL2 provides the following features:

- **Software-defined networking (SDN)**. WL2 uses an SDN-based architecture [23] to achieve scalability in a wide-area L2 network. In each datacenter, WL2 deploys a centralized SDN controller cluster, and creates full-mesh virtual overlay

networks among local hosts (similar to Nicira NVP [22] and OpenStack Neutron [31]). Each controller cluster manages virtual network devices via OpenFlow [26], a protocol allowing the controller to access data plane of OpenFlow-supported switches.

In contrast to solutions like TRILL [35] and OTV [9], WL2 does not require any modifications to existing physical network devices. One goal of WL2 is to introduce minimal change to operational networks in production and allow separate evolution of software and hardware. An important difference to NVP and OpenStack Neutron is that WL2 does not build full-mesh overlays between hosts in all datacenters because of scalability issues and excessive tunnel maintenance overhead. Instead, WL2 sets up gateway nodes at each datacenter and builds full-mesh overlays between gateways.

- **Scalable L2**. With the help of full-mesh overlays in each datacenter and between gateways, WL2 is able to employ a hierarchical addressing scheme and reduce the size of OpenFlow flow tables at each host, hence achieving scalability. The addressing scheme is transparent to cloud end users and enables customizable design scales based on the demands of cloud service providers. A typical design scale of WL2 is tens of datacenters, tens of thousands of hosts per datacenter, and hundreds of VMs per host. In aggregate this translates to millions of VMs. To realize a L2 network abstraction which can cope with this scale, WL2 SDN controller clusters in each datacenter coordinates with each other to handle control-plane traffic such as source learning, address assignment and address resolution, while avoiding all traditional broadcast. Designed for the cloud, WL2 supports multi-tenancy of VMs. WL2's overlay network substrate and hierarchical addressing scheme play a key role in simplifying the design of multi-tenancy and data forwarding.

- **VM live migration** A major feature of WL2 is VM live migration across wide areas between any two datacenters. When a VM is migrated, controller clusters in WL2 coordinate with each other so that all traffic destined for the VM will be redirected to its new location. The traffic redirection includes both private communications between VMs and public communications between the VM and the Internet. Before and after VM live migration, WL2 guarantees that VM IP address has no change and ongoing network sessions of the VM are not disrupted. In particular, WL2 avoids triangular traffic forwarding problem over wide area via efficient flow table updating.

- **High availability**. WL2 provides high availability in the presence of SDN controller failures. Within each datacenter, a WL2 SDN controller cluster consists of multiple controllers running in classic leader-followers mode. Moreover, all critical network data is stored in persistent data storage. With the help of multi-controller support in OpenFlow protocol and idempotence of OpenFlow operations adopted in WL2, we have designed a log-based fail-over mechanism for WL2 controllers. WL2 is able to recover from leader controller failure and resume ongoing network operations with zero control traffic loss. Furthermore, WL2 guarantees high availability of gateways by automatically redirecting traffic to alternate gateways in the presence of failure.

Our implementation of WL2 is deployed and extensively evaluated on a testbed with 2,250 VMs spanning three datacenters interconnected by high-speed networks. We demonstrate that WL2 can support a variety of L2 functionalities at the scale of multi-datacenter, including DHCP, ARP, multi-tenancy, and VM live migration. In our evaluation, WL2 achieves high performance with orders of magnitude reduction in ARP latency compared against traditional L2 network. Moreover, WL2 has zero packet loss when handling ARP traffic, compared to about 50% loss for traditional L2. WL2 has been demonstrated to work well with heavy ARP traffic equivalent to that generated by 62,000 VMs in a typical production environment. In VM live migration over wide areas, WL2 is able to efficiently update OpenFlow flow tables so that network latencies to the migrated VM adapt instantly, with only 1-2 packet losses. We further demonstrate that WL2 is fault tolerant in the presence of both SDN controller and gateway failures.

2. SYSTEM OVERVIEW

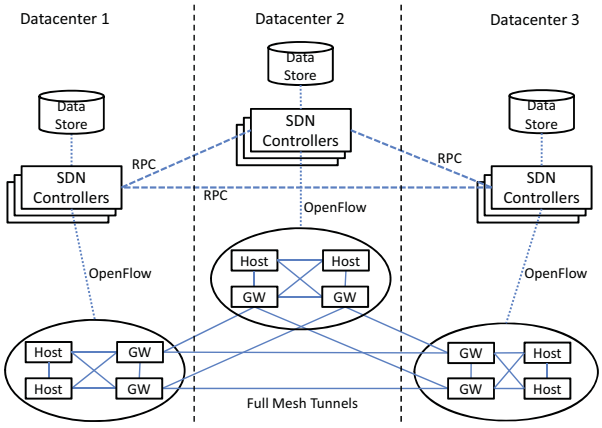


Figure 1: WL2 system overview.

In this section, we present a high-level overview of WL2, introducing its network architecture and the role of SDN controllers in handling control traffic in the network.

2.1 Network Architecture

WL2 is designed to be deployed over multiple datacenters administered by the same entity (*e.g.,* a company or an organization). Datacenters are geographically distributed and communicate with each other through the Internet. They can have arbitrary physical topology (*e.g.,* Fat-tree [12]), and different scales — from tens of thousands of switches to hundreds of thousands. Figure 1 presents an example deployment of WL2 over three datacenters.

Overlay network substrate. Inside a single datacenter, WL2 creates a virtual switch (through OpenvSwitch [32]) in each server (*i.e.,* host), and establishes a logical Ethernet link between every pair of virtual switches through L2 tunneling (*e.g.,* VXLAN [11], GRE [5]), forming a full-mesh overlay network. VMs spawned in a server connect to the local virtual switch to communicate with other VMs. A number of virtual switches in each datacenter are selected as gateway switches (GWs in Figure 1). Gateway switches in different datacenters connect to each other through L2 tunneling on top of the Internet, and form another full-mesh overlay network among themselves. Having multiple gateway switches allows WL2 to support load balancing of inter-datacenter traffic and failover in case a gateway switch is down (Section 3.6). Gateway switches can possess public IPs and serve as Internet routing gateways to enable VMs in WL2 to access the Internet.

Adopting L2 overlay network has several benefits. First, building overlay network leverages the existing mechanisms in the underlying physical network to handle routing, load balancing, and fault tolerance. Next, an overlay network provides a simple abstraction for SDN controllers to operate on, thus reducing controller workload. Also, this approach allows incremental deployment, as it does not require modification to the underlying physical network. There might be concern that an overlay network leads to higher latency and reduced throughput. Study [27] shows that current hardware can support tunneling with negligible loss of performance.

A number of cloud solutions like VMWare/Nicira NVP [22] and OpenStack Neutron [31] also choose full-mesh cloud architecture. In contrast to them, WL2 does not create full-mesh overlay tunnels between all hosts in all datacenters. One major reason is that WL2 is designed for the scale of over hundreds of thousands of hosts spanning multiple datacenters. Full-mesh overlay between all hosts would result in too much overhead in tunnel establishment and maintenance.

SDN Controllers. In each datacenter, WL2 runs a centralized SDN controller (Figure 1) to manage the overlay network substrate, communicating with host and gateway switches through OpenFlow protocol [26]. The controller is responsible for (1) handling L2 broadcast-based control traffic, which is intercepted by the virtual switches and forwarded to the controller (Section 3.2); and (2) setting up/modifying forwarding flow entries in switches to enable/redirect traffic (Section 3.3).

For fault tolerance and high performance, the centralized controller in each datacenter is implemented as a controller cluster consisting of multiple nodes in leader-followers mode (Section 3.7). To achieve high availability, WL2 stores all critical network data in replicated persistent data storage which provides *strong* consistency (*e.g.,* SQL database, ZooKeeper [19]). On the other hand, to speed up data query and minimize load on persistent storage, each controller maintains a local in-memory data copy which is discard-free. When the leader controller fails, follower controllers perform a leader election (*e.g.,* Paxos [24], ZooKeeper [19]) to elect a new leader. Our failover design guarantees no network control traffic loss in face of leader failure (Section 3.7).

Controllers in each datacenter coordinate with each other to create a *logically* centralized controller over all datacenters. Each controller in a datacenter shares control information of the local network with controllers in all other datacenters via message passing, such as Remote Procedure Call (RPC). WL2 employs highly available message queues in each datacenter to buffer messages received from remote controllers, allowing for asynchronous message processing. The distributed data storage across datacenters preserves *eventual* consistency, which tolerates high latency of the Internet across datacenters and is acceptable in a L2 network since slow network data updates in the worst case only delays, rather than prohibits, communications between VMs.

An alternative controller design is to have one *physical* controller cluster managing all datacenters. However, we abandoned this design for an important reason: network latency through the Internet is orders of magnitude larger than within a datacenter. This prevents host and gateway switches in one datacenter from effectively communicating with the centralized controller which resides in another location, in terms of both data query and update.

3. SYSTEM DESIGN

In this section, we elaborate on the design details of WL2. To make WL2 scalable, we have redesigned a number of fundamental L2 control-plane components, including the addressing scheme, DHCP, ARP, the data forwarding scheme, multi-tenancy, and handling of VM live migration. In addition, we present mechanisms that ensure high availability of WL2 controllers and gateway switches against failure, and enhance performance by running controllers of different roles.

3.1 Virtual MAC Address

While the design scale of WL2 is by no means fixed, a typical one we consider accommodates tens of datacenters, tens of thousands of hosts in each datacenter, and hundreds of VMs on each host. This translates to millions of VMs across wide areas. If we naïvely adopt the addressing and forwarding scheme in traditional L2, WL2 will suffer from serious scalability issues. For example, flat addressing in traditional L2 would make each virtual switch, in the worst case, end up with flow entries equal to the number of VMs in the whole network, resulting in state explosion of flow tables. Moreover, multi-tenancy is not well supported in L2. Though VLAN can create different broadcast domains in a L2 network, it only supports at most 4094 domains, a far cry from the requirement of modern cloud providers.

To address these issues, we introduce a tenant-aware hierarchical addressing scheme called Virtual MAC address (VMAC), in place of traditional flat addressing in L2. A VMAC has the same length of 48 bits as a normal MAC address, and consists of four fields as below:

`datacenter_id: switch_id: vm_id: tenant_id`

Here `datacenter_id` denotes the globally unique ID assigned to each datacenter beforehand. Within each datacenter, WL2 assigns a datacenter-wide unique switch ID `switch_id` to all connected virtual switches. `vm_id` differentiates VMs on the same host. The prefix "`datacenter_id: switch_id: vm_id`", in essence, encodes the physical location of a VM and uniquely identifies it in the whole network. The postfix `tenant_id` represents the ID of a cloud tenant allocated by the cloud provider.

Figure 2 shows an example VMAC scheme used in our implementation of WL2 (Section 4). In the scheme, we reserve 7 bits for `datacenter_id`, 16 bits for `switch_id`, 8 bits for `vm_id`, and 16 bits for `tenant_id`. One bit is reserved for multicast address. This permits a deployment of up to 128 datacenters, 65,536 hosts in each datacenter, 256 VMs per virtual switch, and 65,536 cloud tenants. Notice that the length of each field is not static. WL2 allows the cloud provider to adjust the number of bits allocated to each field in VMAC to their needs. For example, major cloud providers (*e.g.,* Amazon EC2) who have a large number of customers can allocate more bits (*e.g.,* 20 bits or 1,048,576) to `tenant_id`, while reducing the size of `datacenter_id` (*e.g.,* 5 bits or 32), based on the number of datacenters they have.

VMAC plays an important role in making WL2 scalable. Below we discuss how VMAC (1) is constructed during VM source learning in Section 3.2, (2) helps reduce flow table size in Section 3.3, and (3) supports large-scale multi-tenancy in Section 3.4.

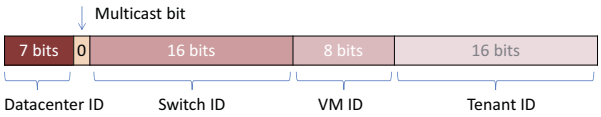


Figure 2: An example VMAC scheme.

3.2 DHCP and ARP

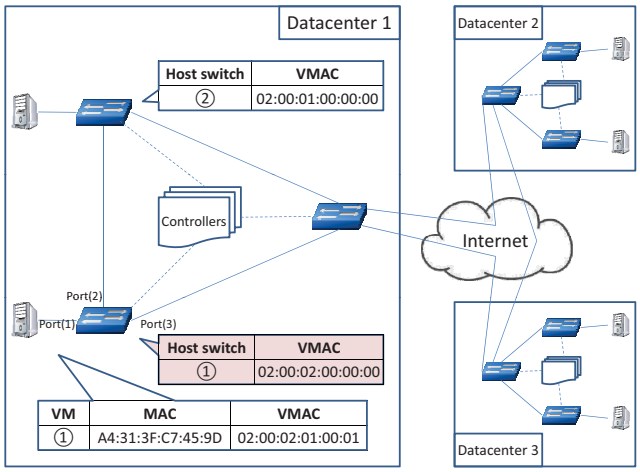


Figure 3: WL2 deployment in three datacenters with MACs and VMACs highlighted.

Pattern Match	Priority	Actions
dl_src=A4:31:3F:C7:45:9D dl_dst=*:00:01	3	GOTO(fwd)
in_port = 1	2	DROP
DEFAULT	1	GOTO(fwd)

Figure 4: Switch ①'s tenant table

Pattern Match	Priority	Actions
dl_dst=04:*	4	PORT(3)
dl_dst=02:00:01:*	3	PORT(2)
dl_dst=02:00:02:01:00:01	2	dl_dst→A4:31:3F:C7:45:9D PORT(1)
DEFAULT	1	DROP

Figure 5: Switch ①'s forwarding table

As in many cloud environments (*e.g.,* OpenStack, Amazon EC2), WL2 uses DHCP to allocate IP addresses to newly spawned VMs. In a traditional L2 network, a node broadcasts a DHCP request packet to obtain its IP address from DHCP server. This broadcast-based approach incurs too much control traffic in a L2 domain, especially for a large-scale L2 network. By comparison, WL2 controller pre-installs a flow entry in each virtual switch, commanding the switch to intercept all DHCP packets (*i.e.,* DHCP DISCOVER and DHCP REQUEST) it receives, and forward the packets to the controller. The controller can serve either as a DHCP agent — forward DHCP packets to DHCP server(s) whose position is known beforehand — or as a DHCP server itself. In either way, it forwards/replies DHCP OFFER and DHCP ACK back to the requesting VM.

WL2 is able to obtain location information of a VM through any control-plane message originated from that VM, such as a DHCP packet. Combining the location information with the tenant membership, WL2 can construct a unique VMAC address for each VM. The VMAC information is globally shared among all controllers through RPC, to support efficient query during address resolution.

Traditional L2 network uses Address Resolution Protocol (ARP) to learn the destination MAC address and initiate communications. Unfortunately, ARP also relies on broadcast and hence has the same scalability issue with DHCP. To avoid ARP broadcast, WL2 installs a flow entry on each host switch, intercepting all broadcast ARP requests and forwarding them to the controller. The controller, on receiving the ARP request, looks up its in-memory ARP table, finds the corresponding VMAC of the requested VM, and replies the VMAC to the requesting VM. The requesting VM sends data packets to the requested VM using replied VMAC as the destination MAC address.

3.3 Hierarchical Data Forwarding

Now we illustrate in detail how hierarchical addressing scheme of VMAC improves scalability of our system, compared to traditional flat addressing scheme. The main insight of VMAC is that it enables WL2 virtual switches to carry out data forwarding using prefix matching. Although the hierarchical addressing scheme can dramatically reduce the flow space, without careful design the controller still needs to install four flow entries for each communicating pair of VMs: one outbound entry at the source virtual switch and one inbound entry at the destination virtual switch for each direction. Therefore, it is challenging for the controller to install flow entries efficiently enough to allow instant communication upon VM initialization, especially when multiple VMs simultaneously try to send traffic through the network.

We address this problem based on the observation that with hierarchical addressing, forwarding tables of L2 switches no longer need to be set up upon detection of a VM — A VM is only significant in terms of its connected virtual switch, and is hidden from the rest of the L2 network behind the virtual switch. This means we can take a two-step flow table setup: (1) after virtual switches are connected to the controller and before any VM is spawned, the controller sets up inter-datacenter and inter-switch flow entries matching only `datacenter_id` and `switch_id` respectively; (2) whenever a new VM is spawned, the controller only needs to set up one flow entry in its locally connected virtual switch, to distinguish it from other locally connected VMs.

Specifically, WL2 performs data forwarding by installing three types of flow entries on each virtual switch, matching on different portions of a destination VMAC address:

- Inter-datacenter flow: match traffic towards VMs in a different datacenter.
- Inter-switch flow: match traffic towards VMs in the same datacenter, but connected to a different virtual switch.
- Local flow: match traffic towards a locally connected VM on the same virtual switch.

Figure 3 presents an example WL2 deployment in three datacenters, each with one gateway and two virtual switches. It uses the same VMAC addressing scheme given in the example in Figure 2. Figure 5 further gives the *Forwarding Table* of virtual switch ① in Figure 3. With this example deployment, we explain in detail how a packet arriving at a virtual switch gets forwarded based on different flow priorities in the forwarding table.

Inter-datacenter flow. The packet is first matched against inter-datacenter flows (in Figure 5 the first row with priority 4), with the following format:

dl_dst=<`datacenter_id`>:*, **actions=**(port_to_gateway)

It states that a packet whose destination MAC address (*dl_dst*) is prefixed by `datacenter_id` will be forwarded to a local gateway

switch. The gateway switch has the same matching field, and forwards the packet to its peer gateway in the destination datacenter. Recall that full-mesh overlay is built between datacenter gateways (Section 2). So given a network of N_{dc} datacenters, each virtual switch would set up $(N_{dc}-1)$ inter-datacenter flow entries, one for each neighboring datacenter.

Inter-switch flow. If the packet does not match any inter-datacenter flow, it will be matched against inter-switch flows (in Figure 5 the second row with priority 3), whose format is as follows:

dl_dst=local_dcid:<`switch_id`>:*, **actions=**(port_to_switch)

It states that a packet whose destination MAC address (*dl_dst*) is prefixed by ID of the local datacenter (`local_dcid`) and ID of a specific virtual switch (`switch_id`) should be forwarded to the host switch indicated by `switch_id`. Given a datacenter hosting N_{sw} switches, each virtual switch sets up $(N_{sw}-1)$ inter-switch flow entries, one for each neighboring virtual switch in the same datacenter.

Local flow. If neither of the above two types of flows is matched, the packet will be matched against local flows (in Figure 5 the third row with priority 2). Local flow entries have the following format:

dl_dst=vmac, **actions=**(rewrite→mac & port_to_vm)

It states that a packet with `vmac` as destination address will be rewritten with a new destination address and forwarded to the local target VM. The new destination address is the actual MAC address of the target VM. Each virtual switch with N_{vm} locally connected VMs has N_{vm} local flow entries.

Lastly, packets that do not match any above flow entry are dropped by default (in Figure 5 the fourth row with priority 1).

WL2 data forwarding scheme makes the flow tables of each host scalable. In total each virtual switch has $(N_{dc}+N_{sw}+N_{vm}-1)$ forwarding entries, a huge reduction of flow table size compared to N_{all_vms} under traditional L2 flat addressing, where N_{all_vms} is the number of all VMs in the whole network.

In WL2, inter-datacenter and inter-switch flow entries can be preset before any VM is spawned. As a result, adding/removal of a VM only incurs the change of a single local flow on a virtual switch, drastically reducing the workload of WL2 controllers when faced with network churn.

3.4 Multi-Tenancy

Multi-tenancy is a crucial feature in a cloud to isolate traffic among different tenants. Virtual LAN (VLAN) in traditional L2 networking, though allowing traffic isolation, supports only up to 4094 tenants, a number far from enough for major cloud providers. WL2, instead, supports large-scale multi-tenancy by assigning a unique tenant ID in the VMAC for each tenant, and filtering inter-tenant traffic through additional flow entries. These flow entries determine if the source and destination of a packet belong to the same tenant, and drops mismatching packets. Though the idea is straightforward, it is challenging to set up appropriate flow entries that both satisfy the traffic isolation requirement and minimize the space complexity. Since OpenFlow does not yet support equality checking between packet fields, we are obliged to enumerate all possible tenant IDs on each virtual switch to allow access between the same tenant. With the large number of supported tenants, this could cause flow table explosion.

We solve the challenge with observation that despite the large number of tenants supported in WL2, the number of tenants on each virtual switch is bounded by the number of ports open to locally connected VMs. Hence, WL2 sets up one more forwarding table on each switch, called *Tenant Table*, to filter traffic from locally connected VMs, before sending it to the forwarding table described in Section 3.3). We reuse the example in Figure 3 to explain the handling of a packet through *Tenant Table*. Figure 4 shows an example table for switch ①. Each *Tenant Table* contains two types of flow entries:

Tenant-forward flow entry. A newly incoming packet is first matched against tenant-forward flow entries (the first row with priority 3 in Figure 4). This type of flow entry allows legitimate intra-tenant packets to get forwarded. The format of a typical entry is as follows:

dl_src=src_mac, **dl_dst=***:`tenant_id`, **actions=**(GOTO(fwd))

It states that if the packet is sent from a locally connected VM (`src_mac`) to another VM of tenant `tenant_id`, it will be forwarded as described in Section 3.3. `src_mac` is supposed to belong to `tenant_id`, and the information is known by the controller when it sets up the flow entry. `GOTO(fwd)` means that the packet will be matched further against the forwarding table `fwd`.

Port-filter flow entry. If the packet does not match any of tenant-forward flow entries, it will be next matched against port-filter flow entries (the second row with priority 2 in Figure 4). The purpose of a port-filter flow entry is to prevent illegitimate local packets to be matched against the default flow entry below and forwarded wrongly. They also prevent any malicious VM who hides from WL2 controller from sending traffic into the network. A port-filter flow entry takes on the format:

in_port=local_port, **actions=**(DROP)

It states that packets coming from any local port are dropped.

Default flow entry. Lastly, packets matching none of the above two types of flow entries are matched and forwarded by default entries (the third row with priority 1 in Figure 4). These packets are incoming traffic sent by other virtual switches.

In summary, a virtual switch with N_{vm} VMs and N_{port} local ports will set up $(N_{vm}+N_{port}+1)$ flow entries in its tenant table. The number of flows is significantly smaller compared to the naïve solution which matches on the source and the destination `tenant_id`.

3.5 VM Live Migration

VM live migration [17] is an important technique in a cloud for dynamic computing resources allocation. For instance, it enables cloud providers to upgrade or replace their hardware devices without affecting any customer's ongoing workload [7].

The challenge of supporting live migration is how to enable fast traffic adjustment after a VM is migrated. Traditionally, the migrated VM will broadcast a Gratuitous ARP (GARP) to the whole network domain after migration. All switches receiving the GARP will change their forwarding table to reflect the new route. In a multi-datacenter scenario, since broadcast does not scale, other mechanisms are needed to adjust forwarding tables on switches.

WL2 supports live migration by maintaining a VM's private access and public access after it is migrated. To properly adjust private access, the controller clusters coordinate with each other, adjusting flow entries collectively so that traffic is redirected correctly and efficiently towards the migrated VM.

3.5.1 Private Access

We focus on traffic adjustment of inter-datacenter VM migration (intra-datacenter migration is simpler). Figure 6 and Figure 7

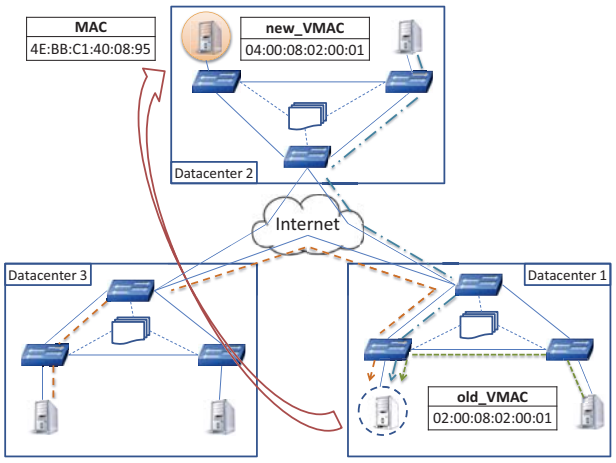


Figure 6: Live migration: A VM has just migrated from Datacenter 1 to Datacenter 2.

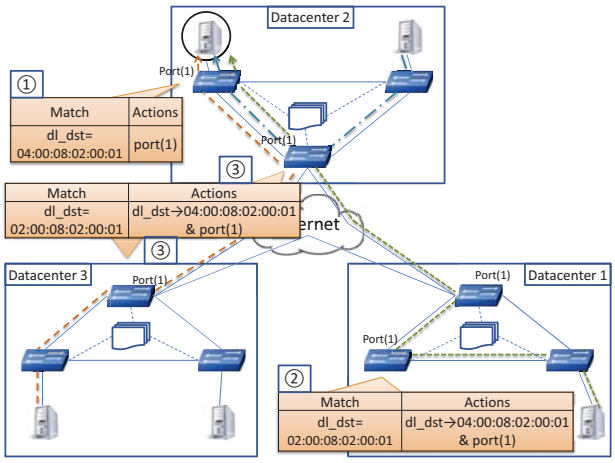


Figure 7: Live migration: All flow tables are updated.

illustrate how live migration is handled in a three-datacenter scenario. In Figure 6, a VM of MAC address "4E:BB:C1:40:08:95" in Datacenter 1 is about to be live migrated to Datacenter 2 (red arrow). The VMAC before the migration is "02:00:08:02:00:01" and is "04:00:08:02:00:01" after live migration respectively. Before live migration, there was a VM in each datacenter communicating with the migrated VM, whose traffic is represented as dashed lines.

After the migration is done, the migrated VM would broadcast a Gratuitous ARP message (GARP), which is intercepted and forwarded to the controller cluster in the new datacenter by the virtual switch (Section 3.2). The controller then takes three steps to adjust flow tables in the network (Figure 7):

Add a local flow towards the new VMAC. When the migrated VM starts in the new datacenter, the local controller cluster would create a new VMAC address for it, based on its location and tenant information, and add a local flow entry in the virtual switch directly connecting the VM (Step ① in Figure 7). Any VM who obtains the new VMAC of the migrated VM is able to communicate with it immediately.

Redirect local flows towards the old VMAC. VMs who send traffic to the old VMAC of the migrated VM will have their packets dropped, before they flush their ARP cache and request the new VMAC. To solve this issue, WL2 instructs the virtual switch which connected to the migrated VM before migration to rewrite the destination address of packets from the old VMAC into the new VMAC, and forward them based on the new VMAC (Step ② in Figure 7).

The above two steps is enough to maintain connection between the migrated VM and all communicating VMs. Also, these two steps are sufficient for intra-datacenter VM migration, in which case the source and destination datacenter happen to the be same. To optimize performance in multiple-datacenter scenario, however, the third step is desirable:

Avoid unnecessary Internet traffic. WL2 optimizes two types of traffic: (1) VMs in the datacenter hosting the migrated VM should be able to contact the migrated VM directly, without having their traffic traverse the Internet twice (*i.e.,* back and forth), as indicated by the above adjustment. Since the number of such VMs, in the worst case, is equal to the total number of VMs in the datacenter, WL2, rather than modifying flow entries in all involved virtual switches, instructs the gateway switch to rewrite the destination address of the outgoing packets from the old VMAC to the new VMAC, and to bounce them back to the migrated VM (upper Step ③ in Figure 7). (2) VMs in other datacenters, excluding the two datacenters involved in migration, will also have their traffic traverse the Internet twice (*i.e.,* triangular traffic forwarding). WL2 instructs the gateways in *all* these datacenters to rewrite the destination address of outgoing packets from the old VMAC to the new VMAC, and directs them to the new datacenter hosting the migrated VM (lower Step ③ in Figure 7). In our implementation, the controller in the datacenter hosting the migrated VM will send asynchronous RPC messages to remote controllers in other datacenters, so that these remote controllers can instruct their local gateways to perform the rewriting.

Dashed lines in Figure 7 show the updated traffic paths after live migration. After a VM is migrated, the controller will in addition send a gratuitous ARP to all VMs which communicate with the migrated VM, informing them of the new VMAC and invalidating the old one. In our implementation, for each VM, WL2 controller keeps track of all VMs that request its VMAC address in the past 60 seconds — the time of normal ARP cache timeout. Hence, all redirecting flow entries setup in Step 2 and 3 can safely timeout after the same amount of time with ARP cache.

3.5.2 Public Access

A VM migrating inside a datacenter does not need to worry about public access, as it preserves the same network gateway and public IP. For a VM migrating between datacenters, WL2 installs a flow entry in its newly connected virtual switch, rewriting the destination of packets which were sent to the old gateway into VMAC of the new gateway, until the VM obtains the address of the new gateway through DHCP.

Floating IP is a popular technique to grant public access to VMs from the Internet in modern cloud (*e.g.,* OpenStack and Amazon EC2 [13]). A floating IP is the public IP of a virtual switch, associated to VMs via Network Address Translation (NAT). After migration, the VM is re-assigned the same floating IP from the newly connected virtual switch, and WL2 would announce a BGP advertisement from within the datacenter hosting the migrated VM, with a short prefix (*e.g.,* `/32`) so that users from the Internet can access the VM without detouring. To avoid too many individual BGP ad-

vertisement updates, it is preferable to migrate all VMs in the same service together so that their BGP advertisement updates can be grouped under a longer prefix.

3.6 Multiple Gateways

In each datacenter, it is desirable to have more than one gateway switch for two reasons. First, as the volume of inter-datacenter and Internet traffic grows, a single gateway is insufficient to handle all the traffic. WL2 can run multiple gateways and balance load among them. The load balancing can be switch-based, flow-based, or remote-datacenter-based, with support of myriad scheduling algorithms (*e.g.,* round robin, weighted). The second reason is to achieve high availability. WL2 sets up a backup node for each gateway switch in the mode of *master-backup*. Each pair of master and backup gateways shares two virtual IPs (VIPs), one public (denoted as `public_vip`) and one private (denoted as `private_vip`) to the local datacenter. VIPs can be maintained via the VRRP protocol [10]. Within a local datacenter, switches build overlay tunnels to the gateway pair using `private_vip` as the tunnel endpoint. Between peer gateways in different datacenters, their overlay tunnels use `public_vip` as endpoint. WL2 sets up two sets of identical forwarding flows on both the master and the backup gateway. In case of master gateway failure, `public_vip` and `private_vip` are automatically shifted to the backup gateway, which keeps forwarding all ongoing traffic.

3.7 Multiple Controllers

Similar to gateways, high availability and performance are important to WL2 controllers. WL2 achieves these two goals by running multiple controllers as a controller cluster. We discuss how WL2 achieves seamless controller failover with zero control packet loss while providing high performance.

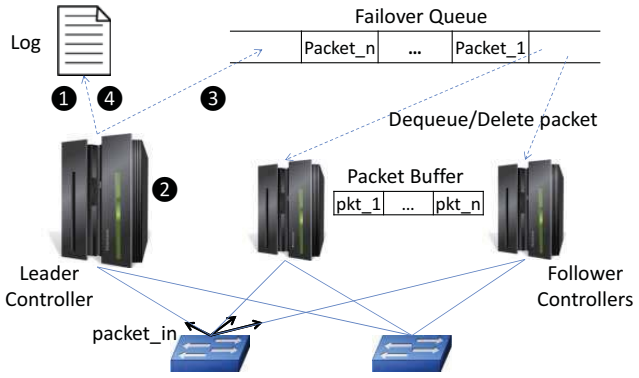
3.7.1 High Availability

To avoid single point of controller failure, WL2 runs multiple controllers, one as leader and others as followers. The controllers share a highly available persistent data backend which stores the network information (*e.g.,* VMACs, address mappings). The storage backend provides strong data consistency. To speed up data lookup, the leader controller maintains an in-memory data copy which is discard-free. Both the leader and followers connect to all switches and receive OpenFlow `packet_in` messages.

Controller operations without failure. Normal controller operations without failure are illustrated in Figure 8. Whenever the leader receives a `packet_in`, it logs the message into persistent storage (Step ①). Then it handles the `packet_in` as described previously, such as VMAC creation and forwarding table setup (Step ②). Afterwards, the leader enqueues the `packet_in` message into a failover queue in the persistent storage (Step ③). Lastly, the leader commits all network data updates into the storage and deletes the log (Step ④). The data updates and log deletion are executed in a transactional style (*i.e.,* all-or-nothing). In particular, the leader takes similar steps in processing RPC requests received from remote datacenters, but not enqueuing packets.

Follower controllers, simultaneously, buffer `packet_in` messages received from switches in local memory. In the meanwhile, they keep dequeuing from the failover queue, which is a broadcast queue to all followers, and delete in-memory `packet_in` message copies that are identical with those retrieved from the failure queue.

In essence, logging in Step ① denotes that the leader begins to process `packet_in`. Log deletion in Step ④ indicates the processing is completed. `packet_in` enqueuing in Step ③ is to synchronize buffered packets between the leader and follower controllers.



Leader Controller Operations			
Step	**Operation**	**Step**	**Operation**
1	Log task	3	Enqueue packet
2	Handle packet_in	4	Write back data + Delete Log

Figure 8: WL2 controller operations.

Controller operations upon failure. When the leader fails, followers perform a leader election [24] to produce a new leader. The new leader would take over and resume unfinished work. First, it reads all network data (*e.g.,* ARP mapping, VMAC registration) from the storage backend. Second, it dequeues all messages from the failover queue, and deletes corresponding `packet_in` messages buffered in the memory. Afterwards, the new leader checks whether any log is left by the previous leader. If the log exists, it performs Step ②∼④ to resume operations relevant to the `packet_in` stored in the log. Otherwise, it does nothing. Lastly, the new leader handles the remaining `packet_in` messages in memory normally as if no failure had ever happened.

Our strategy for controller failover is transparent to the network, with zero control packet loss. We prove that it can recover from leader controller failure at any point in processing a `packet_in`. There are four possible failure points during packet processing, and we will analyze the handling of them by case study. First, failure before Step ① or after Step ④ does not require the new leader to do any failover. Next, failure between Step ① and Step ④ will leave a log entry indicating that the processing of `packet_in` is not finished by the previous leader. The new leader will check the existence of this log and simply re-process the packet by doing Step ②∼④. We note that Step ①, ③, and ④ are *atomic* data write to the storage backend since each is executed in a transactional style. Step ② is *idempotent*, because all OpenFlow instructions used in WL2 (*e.g.,* flow_mod, Packet_out) are idempotent. The new leader can safely execute duplicate OpenFlow instructions without introducing inconsistency. Multiple enqueuings in Step ③ will be ignored by other followers.

3.7.2 High Performance

One observation in L2 network is that packets like ARP requests account for a large portion of control traffic compared to other packets (*e.g.,* DHCP). In WL2, processing these packets only involves data queries (read-only) in the controllers. As a result, to increase the controller performance, one idea is to set aside read-only controllers which are dedicated to processing read-only control packets like ARP requests. Correspondingly, we divide WL2 controllers into two roles, `write-only` and `read-only`.

`write-only` controllers perform normal operations as described previously, including updating forwarding tables and network information in persistent data storage. `write-only` controllers ignore packets like ARP requests. `read-only` controllers are dedicated to processing packets without updating any backend data. They synchronize with persistent storage to get instant updates from `write-only` controllers.

To improve overall performance, `read-only` controllers can be deployed as many as possible. Each switch is connected to all `write-only` controllers and at least one `read-only` controller. To balance load on `read-only` controllers, each switch can connect to them randomly.

4. IMPLEMENTATION

We have implemented a prototype of WL2 in 2000 lines of Python code. WL2 uses the open-source SDN platform Ryu (release 3.4) [2]. Our implementation includes VMAC addressing scheme, DHCP, ARP, data forwarding scheme, multi-tenancy, VM live migration, multiple gateways, and controller high availability and high performance presented in Section 3. Our implementation is open-source released at `github.com/att/ryu`.

WL2 host and gateway switches are created using Open vSwitch (release 2.0.1) [32], a software switch for network virtualization. The intra- and inter-datacenter full-mesh overlay networks are created as VXLAN [11] tunnels. The software switches are configured to *fail-secure* mode and communicate with WL2 controller clusters through OpenFlow protocol (release 1.2) [26]. In the fail-secure mode, a switch drops all packets destined to the controllers if its TCP connection to the controllers is lost.

We use ZooKeeper [19] as the persistent data storage backend of WL2. ZooKeeper serves like a distributed transactional database and provides strong data consistency. Moreover, we leverage the distributed coordination services provided by ZooKeeper to implement task logging, failover queue and leader election [24] used in controller high availability. ZooKeeper can also be used to provide the asynchronous RPC queue for communications among controllers in different datacenters.

5. EVALUATION

We perform extensive evaluation of WL2, including the performance of ARP, traffic redirection during VM live migration, and high availability of gateways and controllers.

5.1 Testbed Setup

To evaluate WL2, we set up a testbed spanning three datacenters, namely `DC1` and `DC2` in New Jersey, and `DC3` in New York. Each datacenter has a private cloud managed by OpenStack [31]. We use OpenStack VMs (KVM as hypervisor) as WL2's host and gateway switches. In each host, we launch nested VMs [25, 15] via either KVM or Docker/LXC [4]. We use nested virtualization because it substantially facilitates our evaluation. It not only saves the overhead of managing physical machines but also enables quick testbed building via VM snapshots, while being transparent to other tenants of our OpenStack private clouds.

In each datacenter, we launch three OpenStack VMs as host switches. Each host has 16 virtual CPUs (vCPUs), 15GB RAM, and 50GB disk. The underlying physical CPUs are Intel(R) Xeon(R) CPU E5-2470 2.30GHz. In addition, we launch one OpenStack VM per datacenter as the gateway switch. It serves as both DHCP server and Internet gateway. Each gateway is allocated 2 vCPUs, 4GB RAM and 10GB disk. We ensure that all hosts and gateways are created on different physical machines so that communications between them are real network packets rather than memory copy on the same physical machine. Moreover, in each datacenter, we create three OpenStack VMs as WL2 SDN controllers. Each controller has 2 vCPUs, 4GB RAM and 10GB disk. Our ZooKeeper cluster is co-located with the controllers. All our OpenStack VMs run Ubuntu 14.04 64bit Server as OS.

Our OpenStack VMs within each datacenter are attached to a private L2 network provided by 1Gbps physical switch. As described in Section 2, we set up full-mesh (one-hop) VXLAN tunnels between hosts within each datacenter. We measure the performance of VXLAN tunnels in terms of latency and throughput. For one-hop VXLAN, its latency is 0.5 to 1ms and throughput is 890Mbps. This is comparable to the performance of underlying physical L2 network. Two-hop VXLAN has much worse performance, with latency of 1.5ms and throughput of 700Mbps. This justifies the full-mesh overlay network substrate adopted by WL2. For better VXLAN performance, NICs with hardware VXLAN support (*e.g.,* [3]) can be used.

Table 1 gives the wide-area network latency and throughput between our three datacenters. We note that the throughput between `DC1` and `DC2` is significantly higher due to a dedicated backbone line between them.

Datacenters	Latency	Throughput
DC1, DC2	3ms	930Mbps
DC1, DC3	4ms	80Mbps
DC2, DC3	5ms	80Mbps

Table 1: Latency and throughput between three datacenters.

Between the three gateway switches in `DC1`, `DC2` and `DC3`, we set up full-mesh VXLAN tunnels. Table 2 summarizes the latency and throughput over one-hop VXLAN on gateways. We observe that wide-area one-hop VXLAN does not affect latency and only adds marginal overhead to network throughput.

Datacenters	Latency	Throughput
DC1, DC2	3ms	825Mbps
DC1, DC3	4ms	75Mbps
DC2, DC3	5ms	72Mbps

Table 2: Latency and throughput between three datacenters over one-hop VXLAN on gateways.

We note that due to the limitation of physical resources, our deployment consists of three datacenters in relatively close geographical regions. However, the testbed suffices our purpose of evaluation in this paper regarding functionality and scalability. WL2 is designed to accommodate much wider geographical regions. We leave deployment of larger scale as future work.

5.2 ARP

ARP latency and loss rate are two basic metrics to measure L2 network performance. ARP latency is the time interval between a node sends out an ARP request and receives the corresponding ARP reply. ARP loss rate is the percentage of ARP requests whose ARP replies are never received. To evaluate the performance of ARP in WL2, we launch nested Docker/LXC container VMs to build a large scale network environment. Light-weight container technology enables efficient VM creation. On each host, we launch 250 nested VMs, leading to 2,250 VMs in total across three dat-

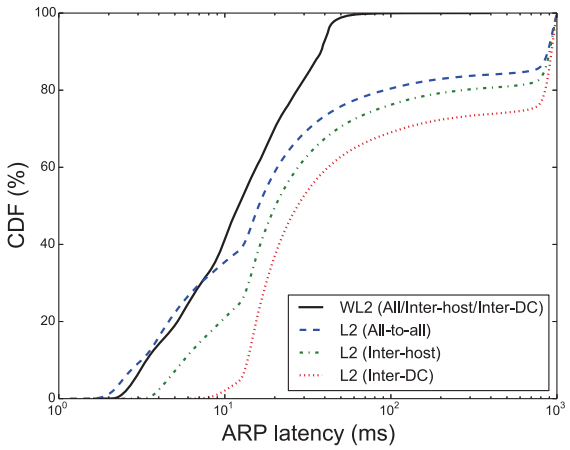


Figure 9: ARP latency in traditional L2 network and WL2.

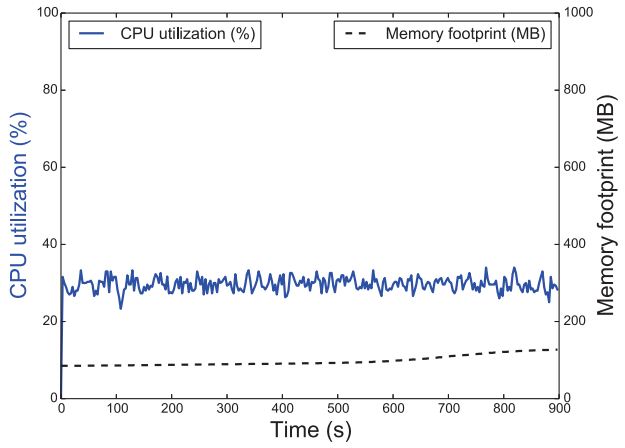


Figure 10: WL2 controller overhead of CPU utilization and memory footprint.

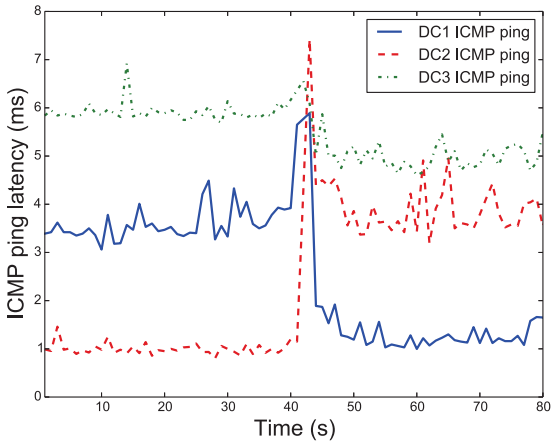


Figure 11: ICMP ping latencies from other datacenters during VM live migration .

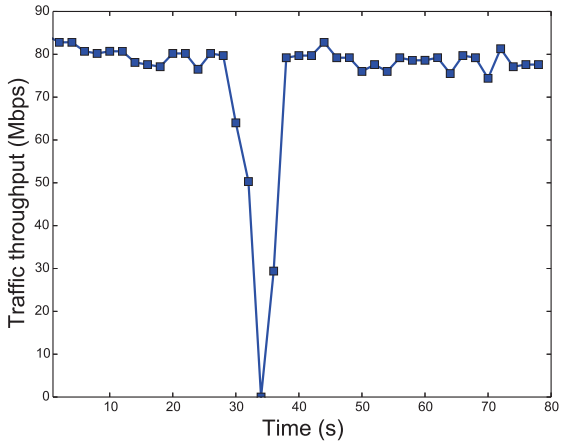


Figure 12: Traffic throughput (TCP) during gateway failure and recovery.

acenters. Our nested VMs run Ubuntu 14.04 64bit Server as OS. After being spawned, each VM obtains an IP address via DHCP, and WL2 sets up one data forwarding flow on its host switch.

We emulate a heavy load of ARP by making each nested VM randomly `arping` each other every 2.5 seconds. We make `arping` bypass local ARP cache and send out ARP request every time. This translates to 900 ARP requests per second in the network. Given a study [29] which shows that a L2 network with 2,456 nodes on average sends 89 ARPs per second, the ARP load in our evaluation is equivalent to that generated by 25,000 VMs in a typical production environment. We run the evaluation for about 15 minutes and compare the results to traditional L2 network provided by Open vSwitch.

Figure 9 shows the performance of ARP latency in both traditional L2 network and WL2. We set `arping` timeout to 1 second. To differentiate the ARP traffic, we divide it into three categories: (1) `All-to-all`: between all VMs; (2) `Inter-host`: between VMs on different hosts; and (3) `Inter-DC`: between VMs in different datacenters. Figure 9 shows that traditional L2 suffers from large ARP latencies, especially for the `Inter-DC` case where an ARP has to be broadcast across wide area to reach its destination and then sent back to the source. WL2, in contrast, achieves much tighter latency distribution under heavy ARP load. In particular, in WL2, 50% ARPs are replied within 10ms, and 80% ARPs are replied within 30ms. Moreover, the three cases of ARP traffic have almost identical distribution in WL2. This is not surprising, as all ARP requests are answered by the local controller cluster in WL2.

Table 3 gives the loss rate of ARP requests. Because a traditional L2 network broadcasts ARPs throughout the network, it leads to heavy control traffic in a large scale network. We note that traditional L2 incurs 44.0%, 49.1% and 54.3% ARP loss rate for `All-to-all`, `Inter-host` and `Inter-DC` respectively. WL2, in contrast, achieves zero ARP loss rate for all cases, guaranteeing good scalability. In addition, we measure the overhead of WL2 in terms of controller CPU utilization and memory footprint. Figure 10 gives the overhead results of the leader SDN controller in `DC2` (follower controllers are idling). It shows a decent 30% CPU utilization (out of a single core) and 100MB memory footprint on

average. The leader controllers in DC1 and DC3 have similar overhead.

With our testbed of 2,250 VMs, we have explored higher ARP load by setting the `arping` interval to 1 second. This translates to 2,250 ARPs per second in the whole network, the same load as if generated by 62,000 VMs in production environment. Under this load, traditional L2 network breaks – most VMs become unreachable even before we can finish starting `arping` on them. However, WL2, with CPU utilization rising to 75%, handles the load well. This demonstrates that WL2 is scalable in handling large-scale wide-area L2 network. To further improve performance and scale, WL2 can deploy `read-only` controllers dedicated to processing ARP requests.

Traffic Type	Traditional L2	WL2
All-to-all	44.0%	0%
Inter-host	49.1%	0%
Inter-DC	54.3%	0%

Table 3: ARP loss rate in traditional L2 network and WL2.

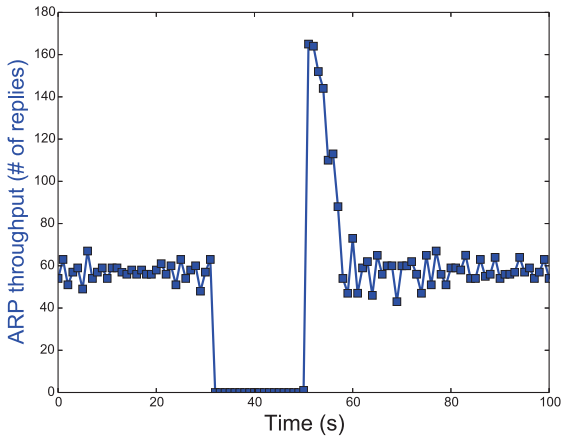


Figure 13: ARP throughput during controller failure and recovery.

5.3 VM Live Migration

We launch nested KVM VMs to evaluate VM live migration in WL2 (Docker/LXC does not support live migration yet). We use NFS as shared storage of nested VMs, so that live migration only involves copying VM memory (256MB) across the network. Live migration with persistent storage [16] is available in KVM and works with WL2 as well, but it takes much longer to copy over the disk.

In the evaluation, we launch a nested VM in DC2, and make it initiate communications to another VM in DC1. We then live migrate the VM from DC2 to DC1, making the two VMs geographically closer. We measure the latency between the two VMs throughout the migration via ICMP ping and plot the results in Figure 11. Live migration starts at time 40 seconds and it takes about 2.5 seconds to finish. Figure 11 shows that before migration, the latency (labeled `DC1 ICMP ping`) is about 3ms, consistent with Table 2. After migration, the latency dramatically drops to 1ms, since the two VMs are now in the same datacenter. We note that the latency jumps to 6ms in the middle of migration. This is because, before WL2 sets gateway flows to redirect the traffic destined to the migrated VM, packets destined to its old VMAC would take a double-way inter-datacenter route – from DC1 to DC2 and then back to DC1 (Section 3.5). We note that this latency spike is only transient and it quickly subsides after gateway flow updates. We also measure the ICMP ping latencies from VMs in other two datacenters towards the migrated VM (marked as `DC2 ICMP ping` and `DC3 ICMP ping`). The results are consistent with inter-datacenter latencies (Table 2), indicating that the flows are correctly updated.

The evaluation results show that WL2 is effective in handling VM live migration across datacenters, by correctly updating forwarding tables. The process is efficient and imposes minimal network disruption. There is only one packet loss of the ICMP pings from DC1 and DC2, and two losses from DC3.

5.4 Gateway High Availability

We evaluate high availability of WL2 gateways by setting up one additional gateway switch as backup in DC3. The two gateways in DC3 pair with each other in master-backup mode, and share public and private VIPs via Keepalived [8]. WL2 sets up identical forwarding flows on them (Section 3.6).

We launch a VM in DC2 sending TCP traffic (using `iperf`) to another VM in DC3. After 30 seconds, we emulate a failure of the master gateway in DC3 by manually powering it off. Keepalived is set to shift VIPs to the backup gateway within three seconds after detecting master failure. Figure 12 shows traffic throughput between the two VMs during gateway failure and recovery. We note that the throughput first drops from 80Mbps to 0, because of master gateway failure. It then quickly recovers to 80Mbps in 10 seconds, after the backup gateway is promoted to master and takes over traffic forwarding. The evaluation shows that gateway failure has minimal network disruption to WL2.

5.5 Controller High Availability

To evaluate WL2 controller's ability to maintain high availability, we use a testbed of 150 nested Docker/LXC container VMs in DC1. We launch 50 VMs on each host switch in DC1. Every nested VM starts random `arping` towards each other every 2.5 seconds. We measure the ARP throughput of WL2, *i.e.,* the number of ARP replies sent by the controllers per second, and plots the results in Figure 13. The ARP throughput is steady at 60/second between time 0 to 30 seconds, when the leader controller is handling control traffic. At 30 seconds, we kill the leader controller process. As a result, the ARP throughput plummets to zero immediately. However, within 20 seconds a new leader is elected and performs failover tasks described in Section 3.7.1, including processing previously buffered ARP requests during failure. The ARP throughput reflects this by jumping to a peak of 165/second, before gradually falling back to a steady volume of 60/second again. In the 20-second failover period, 10 seconds are attributed to Zookeeper leader failure detection, which can be configured to be shorter. During failover all controller operations (*e.g.,* flow installation, VM information registering and sharing) are properly executed, as if the failure had never happened.

6. RELATED WORK

Virtual Private LAN Service (VPLS) extends a L2 network over IP or MPLS networks across wide area. It allows geographically dispersed sites to share an Ethernet broadcast domain. Overlay Transport Virtualization (OTV) [9] extends a L2 network across distributed datacenters with reduced (not eliminated) ARP flood-

ing. Transparent Interconnection of Lots of Links (TRILL) [35] is an IETF protocol that uses L3 link-state routing to create a fairly large L2 network. Compared to WL2's software-defined scalable L2 solution, VPLS has the same scalability issue with L2 network, while TRILL and OTV require dedicated hardware switches.

A number of unconventional designs aim to make L2 networking scalable. SEATTLE [21] creates a global switch-level view by running a link-state routing protocol to build a DHT-based directory. With the directory, SEATTLE eliminates broadcast traffic including ARP and DHCP. VL2 [18] employs a L3 fabric to implement a virtual L2 network targeted for a datacenter with a Clos physical switch topology. It creates tunnels between physical switches and uses a link-state routing protocol to maintain a switch-level topology. PortLand [30] designs a scalable L2 network in a single datacenter with a largely fixed topology (*i.e.,* Fat-tree) of physical switches. It assigns a hierarchical Pseudo MAC address (PMAC) to each node based on its physical location. Though VMAC shares the same design philosophy as PMAC — replacing the flat addressing scheme with the hierarchical addressing scheme to achieve efficient routing and compact forwarding tables — the `datacenter_id` and `tenant_id` in VMAC enable the additional feature of large-scale multi-tenancy in multiple datacenters. Also, the variable length of each field in VMAC offers flexibility when deploying WL2 under different demands. PAST [33] implements a per-address spanning tree routing algorithm to scale Ethernet networks in a single datacenter. [34] shows that software-defined solutions such as NOX are comparable to VL2 and PortLand in a single datacenter. VMWare/Nicira NVP [22] and OpenStack Neutron [31] are two prominent examples of recent software-defined solutions. They build full-mesh overlays within a single datacenter and use an SDN controller to manage the virtual network. Google B4 [20] is a global wide-area network based on a SDN control plane, which translates BGP routing and traffic engineering protocols to underlying OpenFlow operations.

WL2 is inspired by many of these prior works, including overlay substrate, hierarchical addressing, software-defined approach, and dedicated directory system. The biggest difference between WL2 and prior works is that WL2 extends the L2 network abstraction from a single datacenter to multiple datacenters over wide area for the cloud environment. In the context of multi-datacenters and cloud, WL2 addresses a series of unique challenges such as controller cluster coordination, addressing, data forwarding, flow table scalability, and multi-tenancy. Furthermore, WL2 supports a key operation – VM live migration across datacenters.

The global private network of Google Compute Engine [6] shares many features with WL2, including the L2 semantics, multi-tenancy, and wide-area deployment. It also supports intra-datacenter VM live migration, while support for inter-datacenter migration is unclear. However, how Google designs and implements this network is proprietary and unknown.

7. CONCLUSION

In this paper, we have presented WL2, an SDN-based, scalable multi-datacenter L2 network architecture for the cloud. We achieve scalability by re-designing L2 addressing scheme, control-plane protocols, data forwarding mechanism, multi-tenancy, gateway and controller high availability, and VM live migration. With a testbed of real networks and VMs spanning three datacenters, our experiments validate good scalability and fault tolerance of WL2.

8. ACKNOWLEDGMENTS

The authors would like to thank Yun Mao and Xu Chen for their inspiring discussions with us. We also thank the anonymous reviewers for their helpful comments on the earlier versions of the paper. This work is supported in part by CNS-1218066, CNS-1117185, CNS-1117052, CNS-0845552, AFOSR Young Investigator Award FA9550-12-1-0327, AT&T Labs Research summer student internship program and Virtual University Research Initiative (VURI) program.

9. REFERENCES

[1] Amazon Web Services. `http://aws.amazon.com/about-aws/global-infrastructure/`.

[2] Build SDN Agilely: COMPONENT-BASED SOFTWARE DEFINED NETWORKING FRAMEWORK. `http://osrg.github.io/ryu/`.

[3] ConnectX-3 Pro - Mellanox. `https://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-3_Pro_Card_EN.pdf`.

[4] Docker - Build, Ship, and Run Any App, Anywhere. `https://www.docker.com/`.

[5] Generic Routing Encapsulation (GRE). `http://www.ietf.org/rfc/rfc2784.txt`.

[6] Google Compute Engine. `https://developers.google.com/compute/`.

[7] Google Compute Engine. `https://cloud.google.com/compute/docs/robustsystems`.

[8] Keepalived. `http://www.keepalived.org/`.

[9] Overlay Transport Virtualization (OTV) - Cisco. `http://www.cisco.com/c/en/us/solutions/data-center-virtualization/overlay-transport-virtualization-otv/index.html`.

[10] RFC 3768 - Virtual Router Redundancy Protocol (VRRP). `http://tools.ietf.org/html/rfc3768`.

[11] VXLAN A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. `http://tools.ietf.org/html/draft-mahalingam-dutt-dcops-vxlan-03`.

[12] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Seattle, WA, USA, August 17-22, 2008*, pages 63–74, 2008.

[13] Amazon Elastic Compute Cloud (EC2). `http://aws.amazon.com/ec2/`.

[14] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.

[15] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The turtles project: design and implementation of nested virtualization. In *OSDI*, 2010.

[16] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, 2007.

[17] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2-4, 2005, Boston, Massachusetts, USA, Proceedings.*, 2005.

[18] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: A scalable and flexible data center network. In *SIGCOMM*, 2009.

[19] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, 2010.

[20] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. In *SIGCOMM*, 2013.

[21] C. Kim, M. Caesar, and J. Rexford. Floodless in seattle: a scalable ethernet architecture for large enterprises. In *SIGCOMM*, 2008.

[22] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network virtualization in multi-tenant datacenters. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 203–216, Berkeley, CA, USA, 2014. USENIX Association.

[23] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: a distributed control platform for large-scale production networks. In *OSDI*, 2010.

[24] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[25] C. Liu and Y. Mao. Inception: Towards a Nested Cloud Architecture. In *5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2013.

[26] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[27] R. Mehta. VXLAN Performance Evaluation on VMware vSphere 5.1. Technical report, VMware, 2013.

[28] Azure: Microsoft's Cloud Platform. `http://azure.microsoft.com/`.

[29] A. Myers, T. S. E. Ng, and H. Zhang. Rethinking the Service Model: Scaling Ethernet to a Million Nodes. In *HotNets*, 2004OA.

[30] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: A scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM*, 2009.

[31] OpenStack: Open Source Cloud Computing Software. `http://openstack.org/`.

[32] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker. Extending networking into the virtualization layer. In *HotNets*, 2009.

[33] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter. Past: Scalable ethernet for data centers. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 49–60, New York, NY, USA, 2012. ACM.

[34] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker. Applying NOX to the datacenter. In *Eight ACM Workshop on Hot Topics in Networks (HotNets-VIII), HOTNETS '09, New York City*, 2009.

[35] RFC 6326 - Transparent Interconnection of Lots of Links (TRILL) Use of IS-IS. `http://tools.ietf.org/html/rfc6326`, 2011.