SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees*

Chuanxiong Guo[1], Guohan Lu[1], Helen J. Wang[2], Shuang Yang[3]
Chao Kong[4], Peng Sun[5], Wenfei Wu[6], Yongguang Zhang[1]
[1]Microsoft Research Asia, [2]Microsoft Research Redmond, [3]Stanford University
[4]Huawei Technologies, [5]Princeton University, [6]University of Wisconsin-Madison
{chguo,lguohan,helenw,ygz}@microsoft.com, shyang@stanford.edu, chaosfun@gmail.com
pengsun@cs.princeton.edu, wenfeiwu@cs.wisc.edu

ABSTRACT

In this paper, we propose *virtual data center* (VDC) as the unit of resource allocation for multiple tenants in the cloud. VDCs are more desirable than physical data centers because the resources allocated to VDCs can be rapidly adjusted as tenants' needs change. To enable the VDC abstraction, we design a data center network virtualization architecture called SecondNet. SecondNet achieves scalability by distributing all the virtual-to-physical mapping, routing, and bandwidth reservation state in server hypervisors. Its *port-switching* based source routing (PSSR) further makes SecondNet applicable to arbitrary network topologies using commodity servers and switches. SecondNet introduces a centralized VDC allocation algorithm for bandwidth guaranteed virtual to physical mapping. Simulations demonstrate that our VDC allocation achieves high network utilization and low time complexity. Our implementation and experiments show that we can build SecondNet on top of various network topologies, and SecondNet provides bandwidth guarantee and elasticity, as designed.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Packet-switching networks

General Terms

Algorithms, Design

Keywords

Virtual data center, DCN, Bandwidth guarantee

*This work was performed when Shuang, Kong, Peng, and Wenfei were interns at Microsoft Research Asia.

ACM CoNEXT 2010, November 30 – December 3 2010, Philadelphia, USA.

1. INTRODUCTION

With the advent of Amazon EC2, Google AppEngine, and Microsoft Azure, the dream of computing-as-a-utility is becoming a reality [3, 13, 25]. By outsourcing computing to the cloud, businesses and consumers are freed from the cost and burden of planning, purchasing, operating, and maintaining physical hardware and software, and at the mean time, it offers elasticity to meet dynamic demands in resources and good economy with a pay-as-you-go billing model [13].

The Service Level Agreement (SLA) of today's utility computing [3, 25] are centered around computation (dollars per hour per virtual machine or VM), storage (dollars per GB per month), Internet traffic (dollar per GB transferred), and the availability of these resources. Nevertheless, no abstraction or mechanisms and hence no SLAs are available to capture the requirements on the interactions among the allocated VMs, such as bandwidth guarantees among the VMs.

In this paper, we propose *virtual data center* (VDC) as the abstraction for resource allocation. A VDC is defined as a set of VMs with a customer-supplied IP address range and an associated service level agreement (SLA). The SLA specifies not only computation and storage requirements, but also bandwidth requirements for the VMs. The bandwidth requirement is a key addition and offers the significant benefit of performance predictability. A VDC gives the illusion of a dedicated physical data center. This requires VDCs to be *isolated* from one another in all resource access and usage. A VDC is in fact more desirable than a physical data center because it offers *elasticity* which allows its SLA to be adjusted according to the customer's dynamic demands.

To support VDC, we have designed a data center network virtualization architecture called *SecondNet*. The goals of SecondNet are as follows. The design must be *scalable*. For example, bandwidth reservation state maintenance must scale up to hundreds of thousands of servers and millions of VMs in a data center. It must achieve *high utilization* of the infrastructure network and support *elasticity* when tenants' needs change. Finally, the architecture must be practically *deployable* with commodity servers and switches. Providing bandwidth guarantees while achieving these goals is a key challenge and is the focus of this paper.

Maintaining bandwidth allocation state at switches is prohibitively expensive even if only a small subset of the VMs are communicating with one another (Section 3.2). We address the scalability issue by distributing those state at the hypervisors of servers (which need only handle state for their hosted VMs) and use source routing to encode the route into each packet. Consequently, switches in SecondNet are *stateless*. The hypervisors are responsible for bandwidth policing since they are part of the trusted computing base.

For providing bandwidth guarantees, we leverage a special characteristic of data center networks. That is, a data center network is administered by a single entity, and thereby its network topology and failures within can be obtained. This global view of the network allows a centralized bandwidth allocation together with failure handling, which greatly simplifies the problem. In contrast, significant complexity arises for achieving Integrated Services for the Internet due to the numerous ISPs involved [14].

Nevertheless, even centralized bandwidth allocation poses significant challenges. It is an NP-hard problem. We then designed a low time-complexity heuristic algorithm. In this algorithm, we group neighboring servers into clusters of different sizes. When allocating a VDC, we only search the appropriate clusters instead of the entire physical network, greatly reducing the allocation time. This also leads to bandwidth-efficient VDCs because the servers allocated are close in distance. We then use the efficient min-cost flow algorithm to map VMs onto physical servers and leverage the rich connectivity of the physical networks in path allocation. Our allocation algorithm handles incremental expansion and release of resource usage to support elasticity.

For a practical implementation of source routing in the data center environment, we introduce a *Port-Switching based Source Routing* (PSSR). Since the network topology of a data center network is known, PSSR represents a routing path as a sequence of output ports of switches. PSSR can be readily implemented using the MPLS (multi-protocol label switching) [28] capability in existing commodity switches. SecondNet therefore can be ready deployed on top of any of the recently proposed data center network structure, such as fat-tree [2], VL2[9], DCell [10], and BCube [11].

The simulation results of our VDC algorithm show that we can allocate a 5000-VM VDC in 493 seconds on average in a 100,000-server data center. Moreover, our allocation algorithm achieves high resource utilization. We achieve more than 90% server bandwidth for BCube, fat-tree, and VL2. We have implemented SecondNet with commodity servers and switches. We have constructed a 64-server testbed that supports both BCube and fat-tree. Our experiments show that SecondNet provides service differentiation and bandwidth guarantee, and SecondNet can perform path reallocation in seconds and VM migration in tens of seconds for failure handling and dynamic VDC expansion.

The rest of the paper is organized as follows. We describe our VDC service model in Section 2 and overview our SecondNet architecture in Section 3. We present PSSR and the VDC allocation algorithm in Section 4 and Section 5. We use simulation to study VDC allocation in Section 6 and show implementation and experiment results in Section 7. Section 8 presents related work and Section 9 concludes.

2. SERVICE MODEL

Addressing. For address isolation, every VDC has its own IP address space (possibly supplied by tenants), which may overlap with other VDCs' IP address spaces. VMs within the same VDC can communicate with each other just as they are in the same layer-2 Ethernet. VMs in different VDCs cannot talk with each other by default due to security concern. But if needed, they can communicate through layer-3 gateways. Similarly, VMs in VDCs can communicate with computers in the Internet or other private networks.

Service Types. We enumerate the possible scenarios needed by different tenants and make the case for different VDC service types.

Firstly, some applications desire performance predictability and can benefit from having bandwidth guarantees between VM-pairs. For example, many web services can be divided into three tiers: a frontend Web server tier, a middle application tier for business logic, and a backend database/storage tier. It is desirable to have bandwidth guarantees for the frontend-to-middle and middle-to-backend communications so that such web services can serve their tenants with predictable performance. Also, distributed computing applications, such as those that use MapReduce for data-intensive operations, need to shuffle data among many servers. The execution of such a MapReduce job may be severely delayed by a small number of straggling tasks due to contentions for network bandwidth [8]. Bandwidth guarantees make it possible to predict the execution time of such distributed computing applications and hence know how long a VDC needs to be rented.

Secondly, there are applications, such as background file backup, that do not require bandwidth guarantee. A best effort network service is sufficient for them.

Lastly, there are applications whose detailed traffic patterns cannot be predetermined, but still prefer better than best-effort service. For example, when enterprises move their IT infrastructures into the cloud, they can reserve egress/ingress bandwidths for their Web/email/file servers and assign better than best-effort priority to these services for service differentiation.

Based on these observations, we support a service model of three VDC types. Type-0 service provides guaranteed bandwidth between two VMs, which is analogous to Integrated Service [14]. We also provide the traditional best-effort service without any bandwidth guarantee. Between type-0 and best-effort, we offer a type-1 service that provides local egress/ingress bandwidth reservation for a virtual machine. Our VDC model focuses on bandwidth since network bandwidth is a scarce resource [8]. How to include metrics such as latency into the VDC model is our future work.

From a service differentiation point of view, type-0 provides hard end-to-end bandwidth guarantee. Type-1 provides only last and/or first hop guarantee, but its performance is better than best-effort. We therefore assign type-0 traffic the highest priority followed by type-1 traffic, and best-effort traffic has the lowest priority. We monitor and shape the type-0 and type-1 traffic and ensure that they do not violate their reservations. Low priority traffic can use the network bandwidth reserved by high priority traffic if the reservation is not fully utilized. Hence the hybrid of different service types naturally results in efficient network bandwidth usage.

A VDC's bandwidth requirements can be specified using a set of rules of the format [VDCId, srcVM, dstVM, srcPort, dstPort, protocol]→ servType (bandwidth). For example, [vdc_0,vm_0,vm_1,80,*,TCP]→ type-0 (100Mb/s) specifies that TCP packets from vm_0 to vm_1 with source port 80 in vdc_0 requires a type-0 service with an end-to-end bandwidth guarantee of 100Mb/s. SecondNet needs to reserve the sum of the bandwidth required for all type-0 flows from vm_0 to vm_1. In another example, [vdc_1, vm_2, *, 139, *, TCP] →type-1 (50Mb/s) specifies that all TCP packets from source port 139 of vm_2 requires a type-1 service with a local egress bandwidth guarantee of 50Mb/s at vm_2.

3. SECONDNET OVERVIEW

To support the above service model, we have designed a data center virtualization architecture called SecondNet as illustrated in Fig. 1. SecondNet focuses on bandwidth allocation and leverages server hypervisor technology for CPU, memory, and storage isolation and sharing. It introduces a VDC manager for VDC creation, adjustment, and deletion. VDC manager decides how a VDC is mapped to the physical infrastructure. VDC manager, server hypervisors, and switches form the trusted computing base because they are managed by data center operator. VDC manager manages the servers and switches using a spanning tree (SPT) based signaling channel.

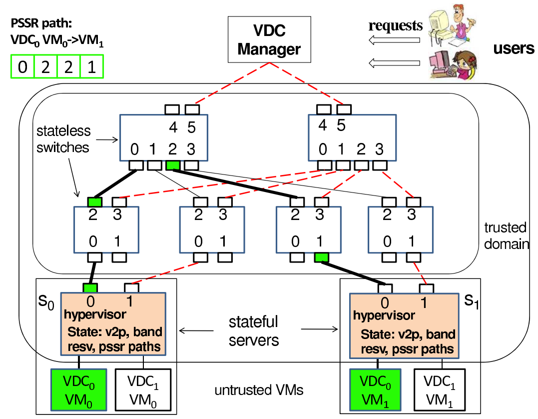


Figure 1: The SecondNet architecture. The red dashed lines form a signaling spanning tree. The black broad lines show a port-switching source routing (PSSR) path.

3.1 VDC Manager

A physical data center is administered by a single entity. This led us to introduce a logically centralized *VDC manager*. VDC manager controls all resources. It performs admission control for VDC requests based on the request SLA and the available physical resources, using a VDC allocation algorithm (Section 5). The allocation algorithm decides how the VMs and virtual edges of a VDC are mapped onto physical servers and routing paths. The algorithm also supports elasticity when tenants expand or shrink the resources of their VDCs, or when various failures happen.

VDC manager assigns every VDC a unique VDC ID and uniquely identifies a VM by its VDC ID and IP address. When VDC manager creates a VM for a VDC, it configures the host server hypervisor with the VDC ID and IP address of the VM, the reserved bandwidths for type-0 and type-1 services, the routing paths for type-0 VM-pairs, and the rule set for mapping traffic to different service types. VMs in a VDC form a conceptual level broadcast domain. Since VDC manager maps VMs to physical servers, it is the natural place for VM-to-physical-server resolution.

VDC manager needs to be scalable and highly fault tolerant. It needs to be up all the time and scale with a large number of VDC requests both in computation and in bandwidth. As we will show in Section 6, one single server can carry out our VDC allocation for VDCs with thousands of VMs at most hundreds of seconds. The traffic between VDC manager and the servers includes VDC creation, adjustment, release requests and the associated configuration messages. The traffic volume is low. For example, the traffic volume for creating a VDC with 1000 VMs is about 30MB, which can be easily handled using the SPT signaling channel.

VDC manager needs to maintain two types of state for its operations. To perform VDC allocation, VDC manager needs to store the whole physical network topology tagged with residual link capacities. For each allocated VDC, VDC manager needs to store all the resource allocation state (i.e., the VM-to-physical-server mapping, egress/ingress bandwidth reservation for type-1 services, and bandwidth reservation and routing paths for type-0 services). Our simulation shows that we need 5GB memory to store all the state for a VL2 [9] network that contains 100k servers. For fault tolerant, consistent, and high available state maintenance, we adopt a similar approach to that of the directory service of VL2 [9] for VDC manager, using replicated state machines and Paxos consensus protocol [23].

3.2 Data Plane

Stateless switches. To provide bandwidth guarantee, we need to pin the routing paths for type-0 VM-pairs. One traditional way for bandwidth reservation is to setup the bandwidth reservation state in not only the physical servers, but also the switches along the routing path. However, this approach incurs severe scalability problem in switch state maintenance. We use VL2 [9] as an example to illustrate the problem. In VL2, a top-of-rack (ToR) switch connects 20 servers, and an Aggregation switch connects 72 ToR switches. Suppose each server hosts 32 VMs and each VM talks to 1000 other VMs. Then the bandwidth reservation state in an Aggregation switch will be 46 million ($32 \times 1000 \times 20 \times 72$) entries. The entries in a server and a ToR switch are 32k (32×1000) and 640k ($32 \times 1000 \times 20$), respectively. The state-of-the-art, high-end switches (e.g., Aristanetworks 7100 [4] and Cisco Nexus 7000 [7]) can only have 16k-128k forwarding entries.

To make state maintenance scalable at switches, we use source routing. With source routing, switches become stateless and are unaware of any VDC and bandwidth reservation state at all. They just perform priority queueing and forward packets based on the source routing information carried in the packet headers.

Hypervisors. Source hypervisors store virtual-to-physical mappings, routing paths and bandwidth reservation state. The number of bandwidth reservation entries in a server is around 32k in the above example. This number can be trivially managed by servers.

Hypervisors classify VM packets to different service types and assign priority to those packets according to SLA. They then monitor and shape the type-0 and type-1 traffic before the traffic enters switches. Best-effort traffic does not need shaping due to its lowest priority. Best-effort traffic therefore can use network bandwidth when type-0 and type-1 services do not fully use their reservations. Hypervisors encode the priority and routing path into packet headers. We note that traffic monitoring, shaping and prioritization must be placed in hypervisors instead of VMs since VMs are not trusted.

Practical deployment. Commodity servers and switches provide the best performance-price tradeoff [5]. We therefore want to implement both priority queueing and source routing on commodity servers and switches. Priority queueing is widely available in both servers and switches. Source routing can be efficiently implemented in current server operating systems as kernel drivers.

However, source routing generally is not available in commodity switches. Furthermore, commodity switches use MAC or IP address for packet forwarding. Some data center network structures may even not use MAC or IP address [10, 11, 15].

To this end, we introduce *port-switching* based source routing (PSSR). Instead of carrying a sequence of next-hop addresses in source routing path, we directly carry the sequence of next-hop output *port* numbers. With PSSR, SecondNet can be implemented with any addressing schemes and network topologies. PSSR can be implemented readily with MPLS (multi-protocol label switching) [28], which is a commodity technology. Fig. 1 shows one PSSR path {0,2,2,1} from vm_0 to vm_1 in VDC_0. Suppose vm_0 in VDC_0 needs to send a packet to its peer vm_1, it first generates a packet that contains vm_1 as the destination address and vm_0 as the source address and delivers the packet to the host hypervisor s_0. The host s_0 then inserts the routing path, {0,2,2,1}, priority, and related information into the packet header and sends the packet to the neighboring switch. The switches then route the packet using PSSR. After the destination server s_1 receives the packet, it removes the PSSR header, and delivers the packet to vm_1.

3.3 Signaling and Failure Handling

VDC manager needs a signaling channel to manage all the server hypervisors network devices. Various server and switch and link failures are inevitable in large data centers. Failures cause network topology changes which then impact both signaling and bandwidth reservation. VDC manager must be notified when failures occur, and routing paths of the affected VDCs must be adjusted. Timely signaling delivery is challenging since the signaling channel itself may fail. In SecondNet, we build a robust, in-band spanning tree (SPT) rooted at the VDC manager as our signaling channel.

In the spanning tree protocol, every device exchanges an SPT message with all its physical neighbors. The message contains the parent and the level of the device. When a device does not know its level, its level is set to NULL. The level of VDC manager is 0. Direct neighbors of VDC manager then get level 1, and so on. A device always chooses the neighbor with the lowest level as its parent. When a device finds that its parent becomes unavailable or the level of its parent becomes NULL, it tries to get a new level from its available neighbors other than its children. As long as the network is connected, the spanning tree can be maintained. Since the spanning tree maintenance message contains parent information, a parent node knows all its children.

VDC manager uses the spanning tree for all VDC management tasks. Devices use the spanning tree to deliver failure messages to VDC manager. VDC manager then adjusts routing paths or reallocate VMs for the affected VDCs if needed. VDC manager also broadcasts the topology changing information to all devices via the spanning tree. Certainly when a link in the spanning tree breaks, the link failure message can only be delivered after the spanning tree has been restored.

We note that the spanning tree is only for signaling purpose hence the traffic volume in the spanning tree is small. We set the priority of the signaling traffic to be the highest. And we can reserve a small amount of the link bandwidth for the spanning tree. Section 6 further shows that the spanning tree converges very quickly even when the link failure rate is 5%.

4. PORT-SWITCHING BASED SOURCE ROUTING

4.1 Source Routing

Since servers know network topology and various failures via the spanning tree, we can remove switches from making routing decisions. This leads us to use source routing for a scalable data plane.

For type-0 traffic, source routing paths are decided by VDC manager. Server hypervisors directly use those paths for routing. For type-1 and best-effort traffic, all the existing DCN routing designs can be easily implemented using source routing at source hypervisors. Both VL2 [9] and BCube [11] use source routing at the server side, hence they can be directly incorporated into the SecondNet framework. In PortLand [15], switches use destination physical MAC (PMAC) hashing to decide the next hop. The source servers can easily calculate the routing path on behalf of the switches in this case. Similarly, the source servers can calculate routing paths for DCell [10], since DCell routing path is derived from DCell IDs.

The overhead of source routing is the routing path carried in the header of every packet. We pay the overhead willingly for a scalable data plane and a flexible routing framework, since the maximum path length of a typical data center network is small (typically 6-8 hops).

4.2 Port-switching

We introduce port-switching to simplify switch functionalities. Traditionally, packet switching is based on destination address. In layer-2 Ethernet switches and layer-3 IP routers, packet switching is based on destination MAC and IP addresses, respectively. Fig. 2(a) shows how layer-2 switching works. When a packet ar-

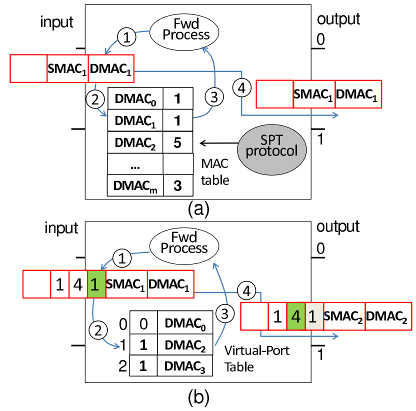


Figure 2: (a) MAC address-based switching. (b) Port-switching.

rives at a port, the forwarding process of the switch extracts the destination MAC address from the packet header (step 1 in Fig. 2(a)) and uses it as the key to lookup the MAC table (step 2). The MAC table contains MAC address in one column and output port number in another. By querying the MAC table, the forwarding process gets the output port (step 3) and forwards the packet to that port (step 4). The MAC table is stored in SRAM or TCAM, and its size must increase accordingly when the network size grows. Further, in order to maintain the MAC table, the switches must run a distributed signaling protocol. IP forwarding works similarly.

Port-switching is much simpler. Instead of carrying MAC or IP addresses, we directly carry the output *port* numbers of the intermediate switches in the packet header. The forwarding process directly gets the forwarding port from the packet header.

Physical port numbers work well for point-to-point links. But a server may have multiple neighbors via a single physical port in topologies such as DCell [10] and BCube [11]. In order to handle this case, we introduce *virtual port*. A physical port can map to multiple virtual ports depending on the number of neighboring servers this physical port connects to. A server maintains a virtual-port table, in which every row represents a neighboring server. The row id corresponds to the virtual port number and each row contains fields including the physical port number and the MAC address of the neighboring server. The size of the virtual-port table is the total number of neighboring servers. The virtual-port table is static in nature unless the neighboring servers change their NICs (which is very unlikely).

Port-switching can be naturally integrated with source routing to form a port-switching based source routing (PSSR), in which a source routing path contains port numbers instead of addresses. Fig. 2(b) shows how PSSR works. Now every packet carries a source routing path identified by output port numbers in its packet header. There is a pointer in the header that points to the next output port number (step 1). The forwarding process uses the next port number to lookup the virtual-port table (step 2), gets the physical port number (step 3), and updates the pointer and forwards the packet through that port (step 4).

PSSR significantly simplifies switch functionalities. Switches are not involved in routing. The virtual-port table is static. The size of virtual-port table is small, since a node typically has at most tens of neighbors. As a comparison, the MAC table (or IP-lookup table) needs at least several thousands entries and its size increases as the network expands.

4.3 MPLS for PSSR

PSSR is easy to implement conceptually - servers encode path and priority information into packet headers, and switches simply perform priority queueing and forward packets based on port-switching. Commodity switches, which are increasingly popular in data centers due to technology advances and the rule of economics of scale [5], can support PSSR as long as it has MPLS, a commonly available switching technology.

In MPLS, switches perform forwarding based on labels carried in packet headers. Labels only have local meaning between two adjacent switches. Switches rewrite the label of a packet hop-by-hop. Labels can also be stacked together to form label stack for MPLS tunneling. In MPLS, labels are established using an LDP (label distribution protocol) signaling protocol.

In SecondNet, we re-interpret MPLS label as port. Consequently, the MPLS label table is interpreted as our virtual-port table. We further implement source routing with MPLS label stack. Since the virtual-port table is static and is pre-configured, signaling protocol like LDP is eliminated. An MPLS label is 20-bits, which is more than enough to describe the number of neighbors a switch or server has (typically less than one hundred). MPLS label also has 3 Exp bits for packet priority. We therefore can implement both PSSR and priority queueing using commodity MPLS switches.

As we have mentioned, VMs in the same VDC form a layer-2 broadcast domain. To support broadcast, we assign a special MPLS tag for each VDC, and use this tag to setup broadcast spanning tree for the VDC in the infrastructure network.

5. VDC ALLOCATION

5.1 Problem Definition

We introduce the notations we will use in Table 1. We denote the physical network as $G(S, X, E)$ where S is the set of servers, X is the set of switches, E is the set of links. Each link has a corresponding link capacity. A server s_i has k_i ($k_i \geq 1$) ports $\{port^j_{s_i} | j \in [0, k_i - 1]\}$.

$G(S, X, E)$	The physical network infrastructure
C_k	Server cluster k
s_i	Physical server i
ib_{s_i}	Residual ingress bandwidth of s_i
eb_{s_i}	Residual egress bandwidth of s_i
$path(s_i, s_j)$	A routing path from server s_i to s_j
VDC_g	Virtual data center with ID g
vm^g_i	Virtual machine i in VDC_g
$r^g_{i,j}$	Requested bandwidth from vm_i to vm_j in VDC_g for type-0 service
er^g_i, ir^g_i	Requested egress, ingress bandwidth for vm_i in VDC_g for type-1 service

Table 1: Notations.

We denote the ingress and egress residual bandwidths of $port^j_{s_i}$ as $ib^j_{s_i}$ and $eb^j_{s_i}$, respectively. We call $ib_{s_i} = \max_j ib^j_{s_i}$ and $eb_{s_i} = \max_j eb^j_{s_i}$ the residual ingress and egress bandwidths, respectively.

For type-0 VDC, we have m virtual machines and the associated $m \times m$ bandwidth requirement matrix R^g, where $r^g_{i,j}$ denotes the bandwidth requirement of the (vm_i,vm_j) virtual edge. The required egress and ingress bandwidths of vm^g_i are therefore $er^g_i = \sum_{j=0}^{m-1} r^g_{i,j}$ and $ir^g_i = \sum_{j=0}^{m-1} r^g_{j,i}$, respectively. For type-1 VDC, we have m virtual machines and the associated egress/ingress bandwidth requirement vector $ER^g = \{(er^g_0, ir^g_0), (er^g_1, ir^g_1), \cdots, (er^g_{m-1}, ir^g_{m-1})\}$.

We can treat best-effort VDC as a special case of type-1 VDC by setting the egress/ingress bandwidth requirement vector to zero. Similarly, we can treat type-1 VDC a special case for type-0 VDC. We therefore focus on type-0 VDC allocation in the rest of this section. We assume one VM maps to one physical server. When a user prefers to allocate several VMs to one physical server, we treat all these VMs as one large VM by summing up their requirements.

The problem of type-0 VDC allocation is to allocate the VMs $\{vm_i | i \in [0, m-1]\}$ to servers s_{π_i} ($i \in [0, m-1]$) selected from the server set S, in a way that the computation requirements (CPU, memory, and disk) of vm_i are satisfied and there exists a path $path(s_{\pi_i}, s_{\pi_j})$ whose residual bandwidth is no smaller than $r^g_{i,j}$ for every VM-pair. In this paper, we use single-path to avoid the out-of-order arrival problem of multi-path.

The VDC allocation problem has two parts: if an allocation exists (decision problem) and if the allocation uses minimal aggregate network bandwidth (optimization problem). The less network bandwidth an allocation uses, the more VDCs we can accept. Both problems are NP-hard. We have proved the NP-hardness by reducing the single-source unsplittable flow [21] to VDC allocation (see [12]).

In the rest of this section, we focus on heuristic design. There are several challenges. First, the algorithm has to be fast even when a VDC has thousands of VMs and the infrastructure has tens to hundreds of thousands servers and switches. Second, the algorithm should well utilize the network bandwidth, and accommodate as many VDCs as possible. Third, the algorithm needs to offer elasticity when tenants' requirements change and timely performs resource reallocation when various failures happen.

Related problems have been studied in virtual network embedding and testbed mapping [6, 30, 27]. The previous solutions cannot be applied to VDC allocation due to the scale of our problem and the VDC elasticity requirement. See Section 8 for detailed discussion.

To our best knowledge, our VDC allocation algorithm is the first attempt that addresses VDC allocation and expansion with thousands of VMs in data centers with hundreds of thousands servers and switches. Furthermore, by taking advantage of VM migration, our algorithm is able to perform bandwidth defragmentation when the total residual bandwidth becomes fragmented.

5.2 The Allocation Algorithm

We pre-configure servers into clusters before any VDC allocation takes place. This is to reduce the problem size and to take server locality into account. There are clusters of different diameters (and hence different sizes). Intuitively, servers within the same ToR switch form a ToR cluster, servers within the same aggregate switch form a Pod cluster, etc. Formally, we use server hop-count, which is the number of hops from one server to another, as the metric to group servers into clusters. A server can belong to multiple clusters, e.g., a 2-hop cluster, a 4-hop cluster, and certainly the whole server set.When the size of a cluster is much larger than that of its belonging small clusters, we combine several smaller ones to form middle size clusters. We denote the clusters as $C_0, C_1, \cdots, C_{t-1}$. A cluster C_k has $|C_k|$ servers. The clusters are sorted in ascending order such that $|C_i| \leq |C_j|$ for $i < j$.

In certain scenarios, users may prefer to allocate VMs to separate locations for reliability reason. In this case, we may use servers at different racks or pods to form clusters. The detail depends on the reliability requirements and are out of the scope of this paper. Though clusters may be formed differently, the VDC allocation procedure is the same.

Fig. 3 shows the *VDCAlloc* algorithm. The input VDC$_g$ has an $m \times m$ bandwidth requirement matrix R^g. The output is m physical servers that will host the virtual machines and the paths set corresponding to R^g. In the first step, we select a cluster C_k. The number of servers of C_k should be larger than the VM numbers in VDC$_g$ (line 2). The aggregate ingress and egress bandwidths of C_k should be larger than those of VDC$_g$ (line 3).

In the second step, we build a bipartite graph with the VMs at the left side and the physical servers of C_k at the right side. We say that a physical machine

```
/*VDC_g has m VMs and an m × m bandwidth matrix R^g.*/
VDCAlloc(VDC_g):
1  for (k = 0;k < t;k + +)/*t is the clusters number*/
2    if (|C_k| < m) continue;
3    if ib(C_k)<ib(VDC_g) or eb(C_k)<eb(VDC_g)
4      continue;
bipartite: /*build weighted bipartite graph*/
5    for (0 ≤ i < m)
6      for (0 ≤ j < |C_k|)
7        if (s_j ∈ C_k is a feasible candidate for vm_i)
8          add edge (vm_i, s_j) to the bipartite;
node_matching:
9    res=MinCostMatching( )
10   if (res== false) continue;
11   for each (i ∈ [0, m − 1]) vm_i → s_{π_i};
path_alloc:
12   fail_flag=0;
13   for each (r^g_{i,j} ≠ 0)
14     if (FindPath(s_{π_i}, s_{π_j}, r_{i,j})==false)
15       fail_flag=1; break;
16   if (fail_flag==0) return succeed;
17 return false; /*fail after trying all the clusters*/
```

Figure 3: The VDC allocation algorithm.

$s_i \in C_k$ is a feasible candidate to a virtual machine vm_j^g if the residual CPU, memory, and disk space of s_i meet the requirement, and the egress and ingress residual bandwidths of s_i are no smaller than er_j^g and ir_j^g, respectively. If server s_i is a feasible candidate to vm_j^g, we draw an edge from vm_j^g to s_i (lines 7-8).

We then use the min-cost network flow [1] to get a matching (line 9). We add a source node *src* at the left side of the VMs and a *dst* node at the right side of the physical servers. We add edges from *src* to the VMs and from the servers to *dst*. We assign weight of an edge as the used bandwidth of the corresponding server. The bipartite matching problem then transforms to the min-cost flow from *src* to *dst* with capacity m. If we cannot find a matching, we continue by choosing another cluster. Otherwise, we go to the third step.

One might assume that different weight assignment policies may result in different mapping result. For example, our weight assignment policy may get better network utilization, since our mapping favors servers with higher residual bandwidth hence more balanced mapping and higher utilization. Our experiment, however, showed that different weight assignment policies have little effect on network utilization. The major reason is because of the clustering heuristic, VDCs will be assigned to appropriate cluster. After that, weight assignment policies cannot significantly affect mapping results and network utilization. In this paper, we simply adhere to our weight assignment policy.

In the third step, we allocate paths for the VM-pairs that have non-zero reserved bandwidths (lines 13-14). We sort the requested bandwidth in descending order and allocate paths sequentially. This is because paths with higher bandwidth request is more difficult to allocate. In the case we cannot allocate path for a VM-pair, we can fail faster and switch to another cluster faster.

We use FindPath to allocate path from s_{π_i} and s_{π_j} with bandwidth requirement $r^g_{i,j}$. In $G(S, X, E)$, we remove the links whose residual bandwidth is smaller than $r^g_{i,j}$, and use shortest-path to get a path from s_{π_i} to s_{π_j}. Since all the links have unit length, we use Breadth First Search (BFS) as the shortest-path algorithm. After we assign a path for a VM-pair, we update the residual bandwidths of the links along the path. If we fail to allocate a path for a VM-pair, we go back to get another cluster and start again. If we do allocate paths for all $r^g_{i,j} \neq 0$, we succeed and return the assigned physical servers and paths. If we cannot find an allocation after searching all the clusters, we fail and reject the VDC allocation request.

VDCAlloc naturally supports VDCs that have multiple service types. For example, when a VM has both type-0 and type-1 requests, a bipartite edge between this VM and a server is feasible only when the egress and ingress residual bandwidths of the server meet the sum of the two requests. After the bipartite is constructed, the rest allocation procedure is the same. VDCAlloc can be executed in parallel for different VDC request as long as they use different clusters. Therefore, a large VDC request will not block a small VDC request. Also during a VDCAlloc, the physical topology may change due to various failures, as long as the related cluster is not affected, VDCAlloc is not affected. Otherwise, we may need to redo the allocation.

The major components, min-cost flow and path allocation, are of low time-complexity. Since all the edges in the bipartite graph have unit weight, MinCostMatching can be solved in $O(n^3 \log(n+m))$, where n is the number of VMs and m is the number of servers in the current cluster. The worst-case time-complexity for path allocation is $O(n^2|E|)$, where $|E|$ is the number of edges of the physical network. The complexity of VDCAlloc certainly depends on how many clusters we need to try before a matching is found. Our simulation (Section 6) shows that even for VDCs with 5000 VMs in data centers with 100k servers, VDCAlloc needs only hundreds of seconds even when network utilization is high.

5.3 VDC Adjustment

VDC has the advantage of dynamic expansion and shrinking as tenants' needs change. VDC shrinking can be trivially performed by releasing the unneeded VMs and bandwidths. VDC expansion, however, is not that easy. There are two expansion cases: increasing bandwidth reservations for existing VM-pairs, or adding new VMs. Also we need to perform VDC reallocation when failures happen. When server failures happen, the hosted VMs disappear. Hence server failures need to be handled by user applications using for example replica which is out of the scope of this paper. But for link or switch failures, SecondNet can perform path reallocation or VM migration for the affected VDCs. It is possible that VDC reallocation may fail. But as we demonstrate in Section 6, VDC reallocation can always succeed when the network utilization is not high.

In this work, we handle incremental expansion and failures with the same algorithm based on VDCAlloc. Our goal is to minimize reallocations of existing VMs. Moreover, we try to reuse existing routing paths. When we increase bandwidth reservation of a VM-pair, we try to increase bandwidth reservation along its existing path. When the existing path cannot meet the requirement (due to link or switch failure, or insufficient bandwidth along that path), we try to allocate a new path for that VM-pair. When path reallocation is not possible, VM migration needs to be performed.

We then maintain a to-be-allocated VM set, which includes the newly added VMs and the VMs that need reallocation. We try to allocate these VMs within the same cluster of the existing VMs using the bipartite matching of Fig. 3. If we find a matching, we allocate paths (step 3 of Fig. 3, with existing paths unchanged). Once we cannot allocate a path between an existing VM and a to-be-allocated VM, we add that existing VM into the to-be-allocated VM set and iterate. If a matching cannot be found, VDC expansion or reallocation within this cluster is not possible. We choose a larger cluster which contains this existing cluster and iterate.

5.4 Bandwidth Defragmentation

An advantage of server virtualization is that VMs can be migrated from one server to another. VM migration can be used for not only server upgrade and maintenance, but also for better network utilization. We use an example to illustrate the idea. Suppose a small number of VMs of VDC_0 are mapped to servers in a cluster C_0 and most of the other VMs are mapped to a cluster C_1. When VMs of some other VDCs in C_1 are released, it is possible to migrate VMs of VDC_0 in C_0 to C_1. The migration not only increases the residual capacity of the physical infrastructure (due to the fact that the inter C_0-C_1 bandwidth of VDC_0 is released), but also improves the performance of VDC_0 by reducing the path lengths among its VMs.

Based on the above observation, we design a VDC defragmentation algorithm as follows. When a VDC is released from a cluster, we check if we get chance to migrate VMs of some VDCs to this cluster. To accelerate VDC selection, we mark VDCs that have VMs scattered in different clusters as defragmentation candidates. A defragmentation is carried out only when the following two conditions are met: 1) the bandwidth reservation of the reallocated VDCs can still be met; 2) the total residual bandwidth of the physical infrastructure is increased. VDC defragmentation is a background process and can be performed when the activity of the to-be-migrated VM is low. Simulation results [12] show that bandwidth defragmentation can significantly improve network utilization.

6. SIMULATIONS

Setup. We use simulation to study the performance of our VDC allocation algorithm. All the experiments are performed on a Dell PE2950 server with 32G memory and 2 quad-core 2.8GHZ Xeon CPUs. We use three typical structures BCube [11], fat-tree [2], and VL2 [9], which represent data center networks of different types and sizes. We did consider tree, but found tree is not suitable for VDC bandwidth guarantee due to its inherent low capacity. For a two-level, 4000 servers tree structure with each ToR gigabit switch connecting 20 servers and an aggregation gigabit switch connecting 200 ToR switches, the aggregation links soon become bottlenecks when we try to allocate several VDCs with 200 VMs.

We also tried to compare our algorithm with several related virtual network embedding algorithms [6, 24]. But the time complexities of the algorithms turned out to be very high. For example, the algorithm in [24] needs 12 seconds to allocate a VDC with 8 VMs in an empty small BCube_2 network with 512 servers. And the algorithm in [6] has even higher time complexity.

The BCube network is a BCube_3 with 4096 servers and 4 layers of 8-port mini-switches (Fig.1 of [11]). The fat-tree has 27,648 servers and three-layers of 48-port switches (Fig.3 of [2]). Links in BCube and fat-tree are 1Gb/s. The VL2 structure (Fig.5 of [9]) has three layers of switches and 103,680 servers. Each ToR switch connects 20 servers with their 1Gb/s ports. A ToR switch connects two aggregate switches with two 10Gb/s ports. The aggregate switches and a layer of intermediate switches form a complete bipartite graph. The aggregate and intermediate switches have 144 10G-ports.

Using the hop-count metric, we divide the servers of the three networks into different clusters. For fat-tree and VL2, these clusters are simply the ToR clusters (2-hop) and Pod clusters (4-hop) etc. For BCube, we get 2048 2-hop clusters, 384 4-hop clusters, 32 6-hop clusters, and one 8-hop clusters.

We define network utilization (*n_util* for abbreviation) as the total bandwidth allocated to VDCs divided by the total link capacity. Similarly, server bandwidth utilization (or *s_util*) is the total server bandwidth allocated to VDCs divided by the total server link capacity.

We use the Google cluster dataset [20] for VDC size distribution. This dataset gives a normalized job size distribution extracted from Google product workloads. The distribution shows more than 51% jobs are the smallest one. But middle size jobs use most of the resources. For example, the 20% middle sized jobs use 65% of the total resources. The probability of large

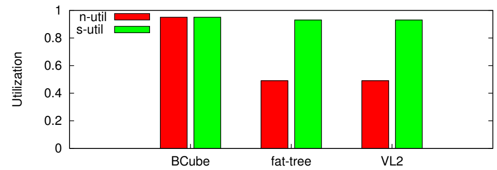


Figure 4: Network and server utilizations for different structures.

jobs are rare. But they use negligible resources. For example, the 0.4% percent largest jobs use 5% resources. We use this dataset to generate synthetic VDC size distribution $[L,H]$, where L and H denote the min and max VDC size.

Utilization. Fig. 4 shows the maximum network and server bandwidth utilizations for the three structures. The VDC size distribution is [10,200]. We add a sequence of randomly generated VDCs into the networks, and get the utilizations when we meet the first rejected VDC. The reported results are mean values for 1000 measurements. We have tested all the three bipartite weight assignment strategies (Section 5.2) and get the same result. The result shows that our VDC allocation algorithm achieves high resource utilization. For fat-tree and VL2, we achieve high server bandwidth utilization (93%) and 49% network utilization. BCube achieves 95% utilization for both *s_util* and *n_util* since all its links directly connect to servers.

The reason that BCube achieves better network utilization than the rest two structures is because all BCube links are equal, which is not the case for fat-tree and VL2. Due to fact that most of the VDCs are small and the locality of VDC allocation, the bisection bandwidth of the high layer switch-switch cannot be fully utilized when the servers run out of bandwidth. BCube therefore accepts more VDCs. The average number of VMs on a server is 20 for BCube, 9.9 for fat-tree and 9.6 for VL2. This is because BCube has larger server bandwidth, which is the bottleneck for fat-tree and VL2. The result shows that BCube performs better for VDC allocation than the rest structures when most of the VDCs are of small size.

Allocation time. Fig. 5 shows the VDC allocation time for the three structures. The VDC size parameters for the three structures are [10,200], [10,1000], and [10,5000], respectively. The results are gotten when the server bandwidth utilizations are 80% (which are close to their max utilizations). The VDC allocation is quite fast even when the server bandwidth utilization is high. For a VDC with 100 VMs in BCube, we only need 2.8 seconds in average. For a VDC with 1000 VMs in fat-tree, we can perform allocation in 20-90 seconds. Even for VDCs with 5000 VMs, we can carry out the allocation within 23 minutes in the worst case. The result shows that the allocation time only grows quadraticly

Link failure rate (%)	Time slot PDF (%)					
	0	1	2	3	4	5
1	62.02	34.14	3.62	0.13	0.09	0
2	61.72	34.74	3.18	0.17	0.12	0.05
3	61.78	34.58	3.38	0.14	0.06	0.04
4	60.38	35.93	3.39	0.17	0.08	0.03
5	59.96	36.22	3.34	0.26	0.18	0.03

Table 2: The distribution of the spanning tree convergence time under different link failure rate for the BCube network.

with the VDC size, which shows the scalability of our allocation algorithm.

Failure handling. We study how the signaling spanning tree reacts to failures. Table 2 shows the convergence time of the spanning tree under different link failure rates for BCube. A time slot is the time to transmit an SPT maintenance message (around 1us for 1Gb/s links). The convergence time is not sensitive to failure rate and the SPT converges quickly. The SPT converges within one time slot with high probability (96%+). SPT therefore builds a robust signaling channel.

We have studied incremental expansion, VDC adjustment due to failures, and VDC defragmentation. See [12] for details. The results show that incremental expansion is much faster than allocation from scratch and that VDC adjustment can be performed by path reallocation or VM migration, and that defragmentation increases network utilization.

7. IMPLEMENTATION AND EXPERIMENTS

We have designed and implemented a SecondNet protocol stack in Windows Server 2008 R2, which integrates Hyper-V hypervisor. In Hyper-V, there is a host OS in the root partition, and VMs are in child partitions. VMs are connected to a software virtual switch. In our implementation, VDCs have different VDC IDs and VMs of different VDCs can have the same private IP address space.

We implement the SecondNet stack as an NDIS (Network Driver Interface Specification) intermediate driver below the virtual switch. The driver maintains a virtual-to-physical table for every VDC, with each entry contains local/peer VM IP, the physical server IP of the peer VM, the reserved bandwidth and PSSR path, and the service rule set. The driver uses a policy manager to map packets into different service types as defined by the SLA rules. It implements leaky bucket for bandwidth regulation for type-0 and type-1 traffic, and priority queueing for traffic differentiation. The driver uses an SPT module for in-band signaling.

The driver is implemented in C and has 35k lines of code. We have prototyped VDC manager using 2k lines of C# and 3k lines of C++ code.

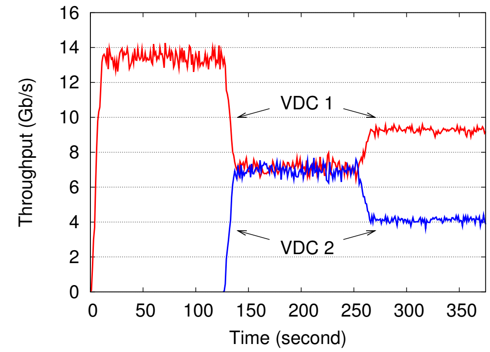


Figure 6: SecondNet provides service differentiation and bandwidth guarantee.

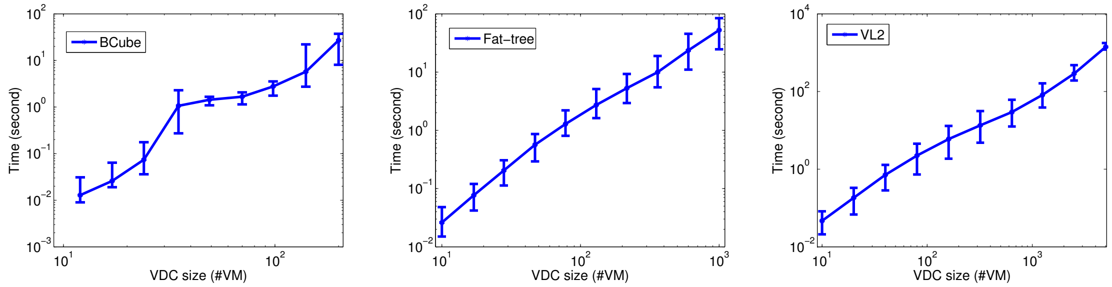
7.1 Testbed

We have built a testbed with 64 servers (40 Dell PE R610 and 24 Dell PE2950), numbered from s_0 to s_{63}. All the servers have four Broadcom Gigabit Ethernet ports and install Windows Server 2008 R2 and our SecondNet driver. We use the first two ports to construct a BCube$_1$ [11] network with 16 8-port DLink DGS-1008D gigabit mini-switches. The BCube network contains 8 BCube$_0$s, and each BCube$_0$ contains 8 servers. We use the third port of the servers and 9 Broadcom BCM956334K MPLS switches (each has 24 GE ports) to form a 2-level fat-tree. The first-level 6 switches use 12 ports to connect to servers and the rest 12 ports to connect to the 3 second-level switches. Each second-level switch acts as 4 6-port virtual switches. Our testbed therefore supports both fat-tree and BCube.

7.2 Experiments

In the first experiment, we use a three-tier Web application to show that SecondNet provides service differentiation and bandwidth guarantee. We use fat-tree for this experiment. We have performed the same experiment using BCube and gotten similar result. We create two VDCs, VDC$_1$ and VDC$_2$, both have 24 VMs divided into frontend, middle, and backend. Each tier has 8 VMs. We map the frontend to s_0-s_7, middle tier to s_8-s_{15}, and backend to s_{16}-s_{23}, and let one server host one VM for each of the VDCs. For each VDC, every VM in the frontend has a TCP connection to every VM in the middle. Similarly, every VM in the middle has one connection to every backend VM. The frontend servers send data to the middle tier, and the middle tier servers send data to the backend. All the routing paths are calculated by our VDC manager to maximize throughput. The two VDCs share the same path set.

Fig. 6 shows the result. In the beginning, only VDC$_1$ has best-effort traffic and achieves around 14Gb/s total throughput. VDC$_2$ starts to generates best-effort traffic at time 127 seconds. Both VDCs get around 7Gb/s. At time 250, we set the traffic of VDC$_1$ to



(a) (b) (c)

Figure 5: The min, mean, and max VDC allocation times. (a) BCube. (b) fat-tree. (c) VL2.

type-0, and set the bandwidth allocation for each TCP connection to 80Mb/s. After that, the total throughput of VDC_1 jumps to 10Gb/s, and the average throughput of TCP connections is 75Mb/s with standard deviation 0.78Mb/s. SecondNet therefore provides bandwidth guarantee for VDC_1 and service differentiation between the two VDCs.

In the second experiment, we show SecondNet well handles link failure and incremental expansion. This experiment uses BCube. We create a VDC with two VMs vm_0 and vm_1, which are hosted at s_0 (BCubeID=00) and s_3 (03). There is a 600Mb/s type-0 bandwidth reservation for (vm_1,vm_0) via path {03,00}. Fig. 7 shows vm_1's aggregate sending rate. At time 62, the level-0 link of s_3 fails. When VDC manager is notified, it re-calculates and adjusts the path to {03,13,10,00} in 77 milliseconds. The interruption time due to link failure is only four seconds.

At time 114, we expand the VDC by adding a new vm_2, and request a 600Mb/s type-0 bandwidth from vm_1 to vm_2. In this case, s_3 cannot meet this new requirement since it has only one link with 400Mb/s available bandwidth. Using the expansion algorithm in Sec 5.3, VDC manager first adds vm_1 to the to-be-allocated VM set, and then migrates vm_1 to s_4(04) and maps vm_2 to s_5(05), and finally allocates path {04,00} for (vm_1, vm_0) and {04,14,15,05} for (vm_1,vm_2). The migration traffic from s_3 to s_4 goes through the path {03,13,14,04} and its throughput is also shown in Fig. 7. The migration transmission finishes in 45 seconds. Note that the interruption time, however, is only five seconds. This is because the VM switches to the new host server only when all its states are synchronized. At time 199, vm_1 starts sending traffic to vm_2, the aggregate throughput of vm_1 becomes 1.2Gbps. This experiment shows that SecondNet well handles both failure and VDC expansion with minimal service interruption time.

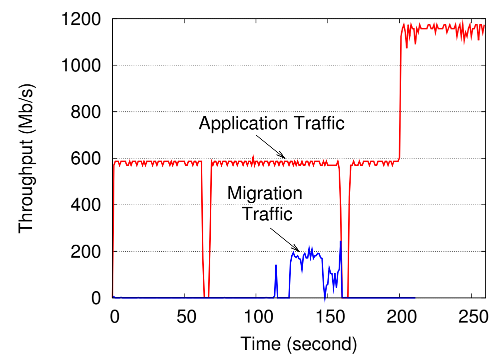


Figure 7: Failure handling and VDC expansion.

8. RELATED WORK

Data center virtualization. Recently, Seawall [29] uses a hypervisor-based framework for bandwidth fair sharing among VM-pairs. It focuses on fair sharing and how resource allocation and bandwidth guarantee can be provided in the framework. NetShare [22] proposes a hierarchical max-min bandwidth allocation. It uses weighted fair queueing for bandwidth allocation among different services, and TCP congestion control or rate limiting to achieve flow level bandwidth sharing within a service. Its relative bandwidth sharing model can be complimentary to the bandwidth guarantee model of SecondNet. FlowVisor [16] is built on top of Openflow [26]. FlowVisor enables different logical networks with different addressing and forwarding mechanisms to share a same physical network. The goal of SecondNet is different from them. SecondNet is end-user oriented and its VDC *hides* all the networking details from end users.

VL2 [9] provides a service model which gives each service the illusion that all the servers allocated to it, and only those servers, are connected by a layer-2 switch. VDC differs from VL2 service in several aspects. 1) A VDC has its own IP address space, whereas a VL2 service is more like an application. 2) We provide bandwidth guarantee for VDCs whereas VL2 cannot. 3) VL2 service model is tightly coupled to their specific network topology, whereas VDC works for arbitrary topology.

Virtual Private Cloud (VPC) [18, 3] has been proposed to connect the cloud and enterprise private networks. VPC does not focus on VMs within a VPC. Amazon provides no implementation details about EC2 and their VPC. Measurement study [19] showed that there is no bandwidth guarantee for EC2 instances.

Virtual network embedding. The virtual network embedding [6, 30] and testbed mapping [27] are related to the VDC allocation problem. In [27], simulated annealing is used for testbed mapping. The work of [27], however, cannot be applied to VDC allocation since it only handles simple physical topology without multipath. Virtual network embedding was studied in [30, 6], with [30] considered path splitting and path migration and [6] used mixed integer programming. The physical networks they studied have only 50-100 nodes. As we have discussed in Section 6, the complexity of these algorithm are high and not applicable to our problem.

Bandwidth guarantee. In the Internet, DiffServ [17] and IntServ [14] are designed to provide service differentiation and bandwidth guarantee, respectively. Compared to DiffServ, SecondNet provides bandwidth guarantee. Compared to IntServ, SecondNet does not need to maintain bandwidth reservation state in switches. SecondNet has the advantages of both DiffServ and IntServ without their shortcomings due to the fact that the network structure is known in advance and data centers are owned and operated by a single entity.

9. CONCLUSION

We have proposed *virtual data center* (VDC) as the resource allocation unit in the cloud, and presented the design, implementation, and evaluation of the SecondNet architecture for VDC support. SecondNet provides VDC isolation, service differentiation, and bandwidth guarantee. SecondNet is scalable by distributing all the virtualization and bandwidth reservation state into servers and keeping switches stateless. Our VDC allocation algorithm achieves high network utilization and has low time complexity. It also enables elasticity by supporting incremental VDC expansion and shrinking. By introducing a port-switching based source routing (PSSR), we have be able to prototype SecondNet using all commodity devices.

10. ACKNOWLEDGEMENT

We thank David Chu, Jim Larus, Sandeep Singhal, Zheng Zhang, Lidong Zhou for their insightful comments and discussions. We thank Zheng Zhang, Feng Zhao, Lidong Zhou for their help and support for testbed construction. We are grateful to Pinyan Lu for his help on the NP-hard proof of the VDC allocation problem.

11. REFERENCES

[1] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows:Theory, Algorithms, and Applications.* Prentice Hall, 1993.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.

[3] Amazon EC2 and VPC. http://aws.amazon.com/ec2 and http://aws.amazon.com/vpc/.

[4] Aristanetworks. 7100 Series 10GBASE-T Data Center Switches. http://www.aristanetworks.com/en/7100T_Datasheet.pdf.

[5] A. Bechtolsheim. The silicon choice for cloud networking, March 2009. http://www.aristanetworks.com/andy/blog-entry/20090326200852.

[6] N. Chowdhury, M. Rahman, and R. Boutaba. Virtual Network Embedding with Coordinated Node and Link Mapping. In *Infocom*, 2009.

[7] Cisco. Cisco Nexus 7000 Series 32-Port 10Gb Ethernet Module, 80Gb Fabric. http://www.cisco.com/.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[9] A. Greenberg et al. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, 2009.

[10] C. Guo et al. DCell: A Scalable and Fault Tolerant Network Structure for Data Centers. In *SIGCOMM*, 2008.

[11] C. Guo et al. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *SIGCOMM*, 2009.

[12] C. Guo et al. SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees. Technical Report MSR-TR-2010-81, MSR, 2010.

[13] M. Armbrust et al. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, EECS University of California at Berkeley, 2009.

[14] R. Braden et al. Resource ReSerVation Protocol (RSVP), Sept 1997. IETF RFC 2205.

[15] R. Mysore et al. PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric. In *SIGCOMM*, 2009.

[16] R. Sherwood et al. FlowVisor: A Network Virtualization Layer. Technical Report Openflow-tr-2009-1, Stanford University, 2009.

[17] S. Blake et al. An Architecture for Differentiated Services, Dec 1998. IETF RFC 2475.

[18] T. Wood et al. The Case for Enterprise-Ready Virtual Private Clouds. In *HotCloud*, 2009.

[19] S. Garfinkel. An Evaluation of Amazon's Grid Computing Services: EC2, S3 and SQS. Technical Report TR-08-07, Harvard University, 2008.

[20] Google. Google Cluster Data. http://code.google.com/p/googleclusterdata/.

[21] S. Kolliopoulos and C. Stein. Improved approximation algorithms for unsplittable flow problems. In *FOCS*, 1997.

[22] T. Lam, S. Radhakrishnan, A. Vahdat, and G. Varghese. NetShare: Virtualizing Data Center Networks across Services. Technical report, UCSD, 2010.

[23] L. Lamport. The Part-time Parliament. *ACM Trans. Computer Systems*, May 1998.

[24] J. Lischka and H. Karl. A virtual network mapping algorithm based on subgraph isomorphism detectionm. In *SIGCOMM VISA Workshop*, 2009.

[25] Microsoft. Windows Azure platform case studies. http://www.microsoft.com/windowsazure/evidence/.

[26] Openflow. http://www.openflowswitch.org.

[27] R. Ricci, C. Alfeld, and J. Lepreau. A Solver for the Network Testbed Mapping Problem. *SIGCOMM CCR*, 33(2), 2003.

[28] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture, Jan 2001. RFC 3031.

[29] A. Shieh, S. Kandula, A. Greenberg, and C. Kim. Seawall: Performance Isolation for Cloud Datacenter Networks. In *HotCloud*, 2010.

[30] M. Yu, Y. Yi, J. Rexford, and M. Chiang. Rethinking virtual network embedding: substrate support for path splitting and migration. *SIGCOMM CCR*, 38(2), 2008.