Walk the Line: Consistent Network Updates with Bandwidth Guarantees

Soudeh Ghorbani and Matthew Caesar
Department of Computer Science
University of Illinois at Urbana-Champaign
201 North Goodwin Avenue
Urbana, Illinois 61801-2302, USA
{ghorban2, caesar}@illinois.edu

Categories and Subject Descriptors

C.2.3 [**Computer Systems Organization**]: Network Operations—*Network management*

Keywords

Consistency, Network Updates, Migration, Software Defined Networking

1. INTRODUCTION

New advances in technologies for high-speed and seamless migration of VMs turns VM migration into a promising and efficient means for load balancing, configuration, power saving, attaining a better resource utilization by reallocating VMs, cost management, etc. in data centers. Despite these numerous benefits, VM migration is still a challenging task for providers, since moving VMs requires update of network state, which consequently could lead to inconsistencies, outages, creation of loops and violations of service level (SLA) agreement requirements. Many applications today like financial services, social networking, recommendation systems, and web search cannot tolerate such problems or degradation of service [5, 12].

On the positive side, the emerging trend of Software Defined Networking (SDN) provides a powerful tool for tackling these challenging problems. In SDN, management applications are run by a logically-centralized controller that directly controls the packet handling functionality of the underlying switches. For example, OpenFlow, a recently proposed mechanism for SDN, provides an API that allows the controller to install rules in switches, process data packets, learn the topology changes, and query traffic counters [13]. The ability to run algorithms in a logically centralized location, and precisely manipulate the forwarding layer of switches creates a new opportunity for *transitioning* the network between two states.

In particular this paper studies the question: given a *starting* network, and a *goal* network, each consisting of a set of switches each with a set of forwarding rules, can we come up with a sequence of OpenFlow instructions, to manipulate the starting network into the goal network, while preserving desired correctness conditions (such as freedom of loops, and bandwidth guarantees)? This problem boils down to solving the following two sub-problems: determining the ordering of VM migrations, which we refer to as the *VM sequence planning problem*; and for each VM that should be migrated, determining the ordering of OpenFlow instructions that should be installed or discarded, which we refer to as the *network sequence planning problem.* One way to address this problem would be to formulate it as an optimization problem, where we compute the order of VM migrations that minimizes the number of bandwidth violations. As we show in Section 2, this formulation results in an NP-hard problem. Since the network needs to be responsive to failures or congestion events that demand immediate reactions, migrations need to be done in real time, and such computational costs is not acceptable. In addition, casual installing/discarding OpenFlow rules while migrating could contribute to problems like transient loops.

To perform the transition while preserving correctness guarantees, we propose an efficient heuristic to "walk" the network between the original and goal topologies. In particular, given the network topology, SLA requirements, and set of VMs that are to be migrated along with their new locations; our algorithm outputs an ordered sequence of VMs to migrate, and a set of forwarding state changes. Our algorithm runs in the SDN controller to orchestrate these changes within the network. To evaluate performance of our design, we simulated its performance using realistic data center and virtual network topologies. We find that for a wide spectrum of workloads, our algorithm performs similarly to the theoretically optimal solution ($< 20\%$ gap), and significantly improves performance compared to a naive approach that randomly selects migrations at each step.

While we believe our core ideas are generalizable to other metrics, in this paper, we focus our discussion on preserving bandwidth guarantees. Preserving bandwidth guarantees is an important problem in its own right. For example, modern data center environments introduce new and stricter requirements.

Motivated by strict latency requirements of applications in data centers [5, 12], importance of predictability of applications performance and tenants cost in the cloud [2, 4] across a wide range of applications, from web applications to

MapReduce-like data intensive applications to HPC and scientific computing applications [2], and the fact that variable network performance is a leading obstacle for predictability [18], increasingly more and more schemes are being proposed to provide tenants *network as a service*, bandwidth guarantees or network resources [2, 3, 8]. To provide stable network performance to applications while migrating, and to prevent violations in SLA requirements that incorporate bandwidth guarantees we formulate the sequence planning problem to impose the restriction that no link capacity can be violated at any point during the migration process. We leave investigation of preserving other SLA requirements (like resilience) to future work.

The remainder of this paper is organized as follows. In Section 2 we formalize the sequence planning problem, and motivate the transition problem by providing examples that demonstrate the need for determining ordering of migrations and forwarding state changes. We further argue why casual OpenFlow rule updates can lead to creation of loops and how such loops can be avoided. In Section 3, we present our algorithm, followed by its performance evaluation. Section 5 concludes the paper.

2. PROBLEM STATEMENT

2.1 VM Sequence Planning

2.1.1 A Simple Example of Sequence Planning

Figure 1 shows a simple example that demonstrates how VM sequence planning can affect the number of possible migrations. In this example, virtual nodes $\{V_1, V_2, ..., V_6\}$ are placed on substrate nodes S1 to S9. Each of the substrate nodes can host at most 1 virtual node. $\{(V_1, V_2), (V_3, V_4), (V_5, V_6)\}$ shows the set of virtual links where all the virtual links have bandwidth requirement 1. All the physical links have bandwidth 1. If our goal is to migrate V_5 from S_8 to S_7, V_4 from S_7 to S_5, and V_2 from S_2 to S_4, then migrating with the sequence $< (V_4, V_2, V_5) >$ would succeed to migrate all 3 specified virtual nodes, while migration with the sequence $< (V_5, V_2, V_4) >$ sequence results in the failure of migration of most of the nodes (2 out of 3).

2.1.2 The Network Model

In this section we give a more formal definition of what we mean by network transition. We assume that there exists a physical or substrate network on which the virtual networks are mapped. We model this underling substrate network by $G_S(V_S, E_S)$, where V_S is the set of all substrate nodes and E_S is the set of all substrate links. We further assume that each node v has a given capacity, represented by $capacity(v)$, that shows the maximum reservable capacity of that node for mapping virtual nodes, e.g., $capacity(v) = 5$ means that node v can host 5 virtual nodes. Also, the maximum reservable bandwidth of each link (u, v) is represented by $b(u, v)$.

Similarly, the set of all virtual nodes and virtual links between them is modeled by $G_V(V_V, E_V)$, where each virtual node v, $v \in V_V$, is mapped to a physical node u, $u \in V_S$, and each virtual link $(u, v) \in E_V$, where $u, v \in V_V$ is mapped to a substrate path $P(s, t)$ where s and t are the substrate nodes that host, respectively, u and v and $P(s, t)$ is a path or sequence of edges between s and t. The bandwidth requirement of the virtual link (u, v) is shown by $r(u, v)$.

$f_N : V_V \to V_S$ and $f_L : E_V \to P_S$ represent, respectively, mapping functions for the assignment of virtual nodes to substrate nodes and virtual links to substrate paths. In this paper, we assume that mapping of virtual links to physical paths is determined by some routing protocol (e.g., shortest paths).

Configuration of the network can be defined by a 3-tuple (G_S, G_V, f_N). We refer to a configuration as a *valid* one, when after mapping the virtual nodes, as determined by f_N, and the virtual links associated with them, f_L, capacity of substrate nodes and their bandwidth constraints are not violated. In this paper, unless stated otherwise, we assume that the initial configuration of network is valid. A *migration set* is a set of 3-tuples (v, s, d) that dictate a virtual node v should be migrated from substrate node s to substrate node d. It should be noted that all the virtual links associated with v should be also migrated. In other words, $\forall u \in V_V$ that $(u, v) \in E_V$, $f_L(u, v)$ should be updated to $P(f_N(u), d)$. Unlike the initial configuration, the final configuration of the network resulting from migrating all the virtual nodes as determined by the migration set might not be always valid. The goal of transitioning is to "walk" the network from initial configuration to a configuration which is as close as possible to the final configuration determined by applying the migration set, while retaining the validity of the configuration, i.e., to migrate the maximum number of virtual nodes without violating node capacities or link bandwidths.

A migration tuple (v, s, d) is called *feasible* if the network configuration stays valid after such operations. A sequence of k migrations is *feasible*, if performing the migrations with the given ordering keeps the network configuration valid at all times. It is *partially feasible* if, iterating through the given sequence, i of those migrations are feasible, where $0 < i < k$; and is infeasible if no migrations can be done by going through that ordering.

A given migration set is *satisfiable* if there exists at least one feasible sequence of migration tuples, is *unsatisfiable* if there exists no such sequence, and *partially satisfiable* when the maximum number of feasible migrations with any possible ordering is smaller than the size of migration set.

With those definitions, the *sequence planning* problem can be defined as determining the ordering of the tuples in a given migration set so as to maximize the number of valid transitions. Simpler cases of sequence planning, like rerouting paths when source and destinations nodes as well as old and new paths are provided, have been previously shown to be NP-hard [10]. Hence, similar to the approach taken in [10] we propose a simple heuristic for sequence planning in Section 3.

2.2 Network Sequence Planning

In the previous section, we presented the sequence planning problem. Determining the sequence of migrations, while challenging, is not the only obstacle encountered for migration in networks. As demonstrated in Figure 2, the order of forwarding state updates in the network could cause transient loops even when a single virtual node is migrated [1]. In this figure virtual node v is migrated from substrate node s to d. An old path P_1 to v which is destined to sub-

[1]Loops can form unless all the old paths and their related forwarding state in all the OpenFlow switches are discarded, before we start establishing new paths by installing new rules in the switches.

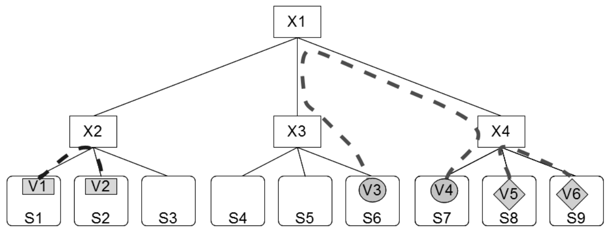

Figure 1: An example showing the effect of sequence planning. Heavy dashed lines show inter-VM communications. The migration goal is to move V_5, V_4, V_2 respectively to S_7, S_5, S_4. Migrating with sequence V_4, V_5, V_2 succeeds to migrate all nodes while migration with sequence V_5, V_2, V_4 can migrate only one node.

strate node s can intersect with a new path P_2 destined to its new location at d at 2 switches X_1 and X_2. If we assume that flow table of X_2 is updated first at time t_1, and after some time X_1 is updated at time t_2, then all the packets destined to v that arrive to X_1 during time interval $[t_1, t_2)$ and the packets that are on the paths between X_1 and X_2 on P_1 will circulate in the loop until t_2 that flow table of X_1 is modified.

The time interval $[t_1, t_2)$ could be significant considering that these two paths (P_1 and P_2) could be any arbitrary paths, therefore P_1 might be torn down (so that we would update X_1) much later than the time that P_2 is set up (which necessitates update of S_2). The situation is exacerbated by the flow install time of the current implementations of OpenFlow controllers: NOX [7], one of the most popular network controllers is claimed to have flow install time of roughly 10ms [21]. Furthermore, the flow install time could highly depend on the specific switch hardware used [17].

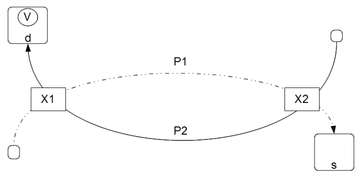

Figure 2: Creation of transient loop due to migration.

To minimize the effect of such loops we propose the following simple fix: whenever a virtual node v is migrated from substrate node s to d, we start updating the state of switches closest to d, by tearing down the forwarding rules matching the v as their destination (that are directed to v's old location) and installing new rules forwarding the packets sent to v to its new location, down to those furthest from it. In the given example, with this scheme, X_1 would be updated at t_1, and X_2 at t_2, and only those packets that are already on P_1 between X_1 and X_2 *at* time instant $t1$ *and* arrive at X_2 after t_2 will just circulate *once* in that loop.

It should be noted that we do not rely on any assumption about the OpenFlow controller to use, and any implementation that fulfills the basic requirements of SDN standards would meet our requirements.

3. OUR APPROACH

3.1 Algorithms

As stated before, we propose a heuristic for sequence planning problem. An algorithm for solving sequence planning takes as input the initial state of the network (i.e. underlying physical topology along with mappings that show the allocation of virtual nodes and links over that topology) and migration set that determines which virtual nodes need to be migrated, as well as sources and destinations of migrations. The algorithm then computes and outputs a sequence for performing migrations.

The naive approach for determining the theoretically best sequence (i.e., a sequence that would result in maximum possible migrations) would be to generate all the possible permutations of members of migration set and output the one with the maximum number of feasible tuples. This is presented in algorithm 1. If we assume that there are n VMs to migrate and it takes c time to test the feasibility of migration of a single node [2], then the optimal algorithm runs in $O(nc \times n!)$. As Section 3.2 shows, this cost becomes prohibitive, even for small instances.

Algorithm 1 Theoretically optimal sequence planning

```
Input: Network configuration $(G_S, G_V, f_N)$ and migration set $\{(v_1, s_1, d_1), ..., (v_k, s_k, d_k)\}$
Output: Migration sequence

permutation set ← all permutations of $k$ tuples ∈ migration set
max score ← $-1$
migration sequence ← $NULL$
for all sequence $s$ ∈ permutation set do
    score($s$) ← number of feasible migrations of $s$

    if score($s$) > max score then
        max score ← score($s$)
        migration sequence ← $s$
    end if

end for

return migration sequence
```

An alternative approach would be to select a random ordering of nodes to migrate (algorithm 2). This reduces this cost to $O(cn)$, but as will be shown in Section 3.2, for a broad spectrum of real scenarios and workloads, it performs poorly in practice, failing to migrate a substantial number of nodes.

We, therefore, propose our own simple heuristic (algorithm 3) that performs near optimal while having significantly lower cost: $O(cn^2)$. Informally, our heuristic iterates over all the nodes in migration set, and

[2]This c value depends on various elements that would contribute to the cost of deallocation of the old paths connected to the node that should be migrated, and calculation and allocation of its new paths, like size and topology of the underlying network, and connections of the node.

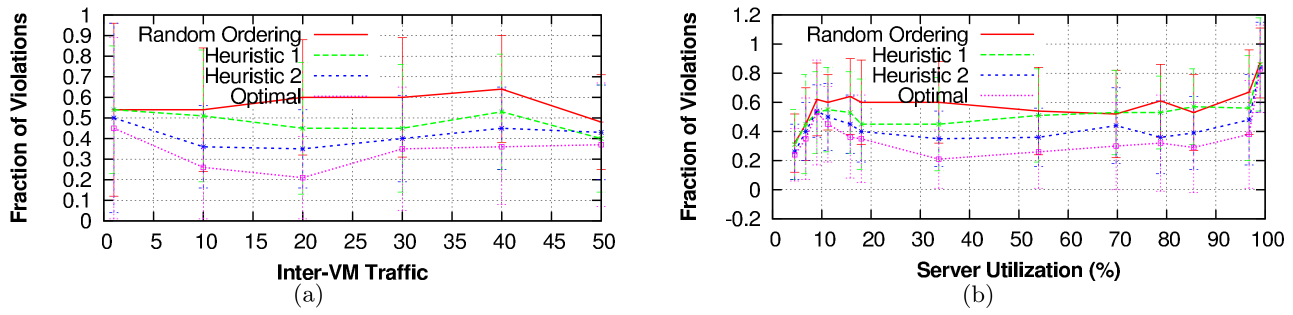

Figure 3: Effect of load on performance of algorithms.

for each tuple (v, s, d), it computes a score that shows how many migrations would be feasible right after migration of node v. It then sorts the tuples according to those scores (descending order) and return the resulted sorted list as the output [3]. In Figure 1, for instance, $score(V_5, S_8, S_7) = 0$, $score(V_4, S_7, S_5) = 2$ and $score(V_2, S_2, S_4) = 0$. Our heuristic, therefore, outputs either of these two sequences (since ties are broken at random): $<(V_4, S_7, S_5), (V_2, S_2, S_4), (V_5, S_8, S_7)>$ or $<(V_4, S_7, S_5), (V_5, S_8, S_7), (V_2, S_2, S_4)>$ that happens to be identical to what optimal algorithm would generate in this case: it migrates all the nodes as specified by the migration set.

Algorithm 2 Random sequence planning

Input: Network configuration (G_S, G_V, f_N) and migration set $\{(v_1, s_1, d_1), ..., (v_k, s_k, d_k)\}$
Output: Migration sequence

```
$migration\ sequence \leftarrow$ random permutation of $migration\ set$

return $migration\ sequence$
```

3.2 Performance Evaluation

3.2.1 Experimental Settings

We evaluate performance of our algorithm using simulations, and test it against two baselines: the optimal solution, and a baseline approach where we select migrations randomly (Algorithm 2), over an extensive set of scenarios and network settings.

Allocating virtual networks on a shared physical network or on a shared physical data center has been extensively studied previously [2, 8, 23]. Both for the physical underlying network and for the VNs, we borrow the topologies and settings used in these works. More specifically, for the underlying topology, we evaluate our algorithms on random graphs, trees, fat-trees, D-Cells, and B-Cubes. For VNs, we use 3-tier graphs, which are common for web service applications, stars, and trees [2, 8, 23]. Furthermore, for initially allocating VNs before migration, we use SecondNet's algorithm [8], because of its low time complexity, achieving high utilization, and support of arbitrary topologies.

We select random virtual nodes to migrate, and randomly select their from the set of all substrate nodes that have sufficient spare capacity. We acknowledge that diverse scenarios for which migrations are performed might have different goals for node or destination selections and such selections might impact the performance of algorithms. We leave exploration of such mechanisms and the performance of our heuristic over them to future work.

Our experiments are performed on a Intel Core i7-2600K machine with 16GB memory.

Algorithm 3 A simple heuristic for sequence planning

Input: Network configuration (G_S, G_V, f_N) and migration set $\{(v_1, s_1, d_1), ..., (v_k, s_k, d_k)\}$
Output: Migration sequence

```
$migration\ sequence \leftarrow$ random ordering of $migration\ set$
for all $t \in migration\ set$ do
    $score(t) \leftarrow 0$
    if $t$ is feasible then
        $score(t) \leftarrow$ number of feasible migrations in $migration\ set$ assuming that migration $t$ is performed
    end if
end for

$migration\ sequence \leftarrow$ sorted $migration\ set$ (descending order of $score$ values)

return $migration\ sequence$
```

3.2.2 Results

For testing the algorithms for each physical and VN topology, we first consider the case that utilization is high, i.e., the provider of the substrate network has allocated as many VNs as possible on her network and the network is "saturated". This could happen either because the physical nodes have hosted the maximum number of virtual nodes and new VNs cannot be allocated without violation of capacity constraints of nodes, or because traffic requirements between VMs of already-allocated VNs are high enough to make it infeasible to admit new VNs without violation of link capacities.

Not surprisingly, the performance (as well as the running

[3]After migrating each VM, packets that arrive at the old VM location are dropped. With taking this approach, although we compromise on performance, consistency will be still preserved. We leave investigation of alternative approaches to future work.

time), of our heuristic depends on the number of migration requests that are batched and processed together (i.e., size of the migration set). As the number of migration requests that are collected together and processed at the same time increases, the gap between the performance of our heuristic and randomly planning the sequence increases. Figure 4 depicts this fact for 10 round of experiments over a 200-node tree where each substrate node has capacity 2, substrate links have bandwidth 500M, and VNs are in form of 9-node trees with links with 10M bandwidth requirement. The line labeled *heuristic 1* shows the fraction of *migration set* members that our heuristic fails to migrate without violation of bandwidth guarantees, when the batch size (size of migration set) is set to 5, e.g., for performing 20 migrations that would translate into 4 rounds of running our heuristic each with migration set of size 5, where the initial superset of migrations is randomly partitioned between these 10 sets. The line labeled *heuristic 2* shows the results for the case when their original migration set is not partitioned (and thus all the migration requests are processed at once) [4]. Error bars show standard deviation.

We ran other sets of experiments to better understand the effect of network load over the performance of our algorithm, by varying the bandwidth requirement of VNs, $r(u, v)$s. Figure 3 shows the results of an experiment over a 400-node physical network in the shape of a tree with links with capacity 1G and VNs with 9 nodes in the shape of trees with branching factor 3 (the results were generated by varying the traffic between VMs of each VN). As Figure 3(a) shows for very low and very high values of inter VM communication traffic, migration fails with almost similar rates across all the approaches, including random and optimal. This demonstrates that the performance of the random ordering heuristic is similar to the performance of the computationally-expensive theoretically-optimal solution. This graph shows the spectrum of inter-VM traffic for which the algorithms were tested.

These experiments resulted in various values of network utilization for the underlying network (due to the fact that the number of VNs that could be admitted varied by bandwidth requirements of VNs). They show that when network is saturated this way (it accepts VNs until it can no longer provide either their bandwidth requirement or it has not enough VM capacity left for admitting new VNs), the performance seems to depend on server utilization: for very high ($> 95\%$) and low server utilization ($< 15\%$) the algorithms perform almost similarly, and for the broad range between these two values ($15\% < utilization < 95\%$) there is a considerable gap between optimal and random solutions and our algorithm is close to optimal. We observe similar trend when server utilization is in this range and the network is saturated in the same manner across other topologies, node capacities and link bandwidths for substrate networks and VNs.

[4]Rising trends in fraction of violations for random planning and heuristic 1 are due to the fact that as the number of migrations increases, more and more VMs will be replaced from the original place specified by the allocation algorithm. This sub-optimal allocation of nodes of VNs makes the feasibility of a random migration less likely, e.g., it is more likely to encounter violation while migrating the 10th VM than the 1th. It is interesting to note that even with quite large number of migrations, the fraction of violations encountered by optimal solution and heuristic 2 remains almost constant.

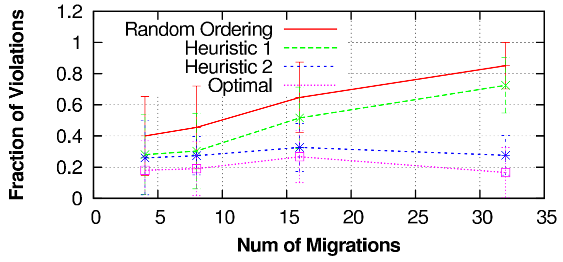

Figure 4: Fraction of migrations that would lead to violation of link capacities with different algorithms.

It should be noted that it is not safe to assume that any of these observations regarding load would hold true for other allocation schemes. E.g., when we manually vary server utilization (leaving the system underutilized despite having free capacity) rather than feeding in VNs till the network is full, (not surprisingly) the success rate of these algorithms are all very close to each other.

As mentioned before, optimal algorithm is considerably computationally expensive. Even when we use some optimization of the algorithm (e.g., returning after encountering the first sequence where all the migration tuples are feasible), and tricks like caching the paths, still for small problem instances like a 100-node tree as the substrate network and 8-node trees as the VNs, 10 runs of the optimal algorithm each took 38.54 *seconds* on average for migrating only 5 nodes (this is just for determining the sequence of 5 nodes to migrate, not performing the actual migration), while our heuristic took 0.37 *seconds*. We observe over different experiments that our heuristic is orders of magnitude faster, and the difference rapidly increases as the number of nodes to migrate increases. This stands to reason, as the asymptotic running time of algorithms in the previous Section. This prohibitive cost of optimal algorithm makes it infeasible to run it over very large problem instances. Therefore, while we have data points comparing our heuristic against random planning for large problem instances which show considerable gaps between them, in order to be able to compare with the theoretically optimal solutions, majority of the results reported in this paper are for the experiments that are conducted over substrate networks with limited number of nodes (< 1000), VNs with 30 or fewer nodes, and migration sets smaller than 100. Note that high time complexity of optimal sequence planning makes it impractical, since data centers are highly dynamic environments and the configuration of the network could dramatically change during the time that optimal planning is being computed. This makes such plans less useful by the time they are computed. Furthermore, many of migration scenarios like decreasing congestion require rapid reaction and cannot tolerate such delays.

4. RELATED WORK

Applications of network migration

Although there has been numerous research proposals that promise improved utilization, security, performance, etc. by leveraging migration of end hosts or network elements [8, 20, 23], or seamless ways for organizations to utilize their private cloud along with public cloud [9, 11], the real deployment of such proposals seem to be impeded by the inevitable disruption in service during migration, as well as the highly congested data center links that make it difficult to migrate without bandwidth violation. Our work, therefore, provides the essential requirements for deploying such schemes in real world by scheduling the migration to reduce bandwidth violation and using the techniques to shield applications from disruption while migrating.

Migration of virtual machines
VM migration is a mature field of research. Recent advances in live VM migration, which enable migration of VMs with the disruption on order of several hundred milliseconds, gives cloud providers and administrators a powerful management tool that facilitates load balancing, data center consolidation and expansion and disaster avoidance [1, 6]. Our work leverages these advances.

Migration of virtual networks
Network element migrations has been previously studied due to its great use in network management. As an instance, VROOM [22] focuses on migration of virtual routers. Its approach for setting up the network before migrating the end-points have been used in [14] where the authors extend it to OpenFlow networks.

Retaining consistency during network updates
There has been much work on retaining desirable properties during network convergence. oFIB instills an ordering over updates distributed over a network to mitigate convergence-related outages [19]. Other work improves consistency of OpenFlow networks, at the cost of increasing state in switches to store duplicate table entries [15, 16]. Neither of these works support bandwidth or other metric guarantees during convergence. Other works [10] describe heuristics for determining the ordering of paths to be rerouted in an MPLS network. The focus of our work is on migration of VMs rather than rerouting of paths.

5. CONCLUSION

Migrating VMs along with their connections has numerous benefits in data centers, ranging from load balancing to power saving to optimization of performance and utilization. However, directly migrating individual components can lead to inconsistencies and overloads of resources. In this paper we showed that by instilling an ordering over changes to the virtual network, we can perform this migration efficiently while providing strong guarantees on the ability to meet constraints on bandwidth and loop-freedom. We show that while this problem is in general NP-hard, a simple and efficient heuristic has near optimal performance across a wide range of topologies and workloads.

We believe our core ideas and techniques are generalizable to other constraints besides bandwidth and loop-freedom such as security properties, network policies, and failure-resilience. In future work, we plan to investigate application of our algorithm to such alternative constraints.

6. REFERENCES

[1] Virtual Machine Mobility with VMware VMotion and Cisco data center interconnect technologies, 2009.

[2] H. Ballani, P. Costa, T. Karagiannis, and A. I. T. Rowstron. Towards predictable datacenter networks. SIGCOMM, 2011.

[3] P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf. NaaS: Network-as-a-Service in the Cloud. Hot-ICE, 2012.

[4] B. Craybrook. Comparing cloud risks and virtualization risks for data center apps, 2011.

[5] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: scaling flow management for high-performance networks. SIGCOMM, 2011.

[6] K. Greene. NTT, in collaboration with Nicira Networks, succeeds in remote datacenter live migration, 2011.

[7] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3), July 2008.

[8] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. SecondNet: A data center network virtualization architecture with bandwidth guarantees. CoNEXT, 2010.

[9] M. Hajjat, X. Sun, Y.-W. E. Sung, D. Maltz, S. Rao, K. Sripanidkulchai, and M. Tawarmalani. Cloudward bound: planning for beneficial migration of enterprise applications to the cloud. SIGCOMM, 2010.

[10] B. G. Jozsa and M. Makai. On the solution of reroute sequence planning problem in mpls networks. *Computer Networks*, 42(2):199 – 210, 2003.

[11] S. Y. Ko, K. Jeon, and R. Morales. The hybrex model for confidentiality and privacy in cloud computing. HotCloud, 2011.

[12] M. Lee, S. Goldberg, R. R. Kompella, and G. Varghese. Fine-grained latency and loss measurements in the presence of reordering. SIGMETRICS, 2011.

[13] 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, Mar. 2008.

[14] P. Pisa, N. Fernandes, H. Carvalho, M. Moreira, M. Campista, L. Costa, and O. Duarte. Openflow and xen-based virtual network migration. In *Communications: Wireless in Developing Countries and Networks of the Future*, volume 327 of *IFIP Advances in Information and Communication Technology*, pages 170–181. Springer Boston, 2010.

[15] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. SIGCOMM, 2012.

[16] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. Consistent updates for software-defined networks: Change you can believe in!". HotNets, 2011.

[17] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. OFLOPS: An Open Framework for OpenFlow Switch Evaluation. PAM, 2012.

[18] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proc. VLDB Endow.*, 3(1-2), Sept. 2010.

[19] M. Shand, S. Bryant, S. Previdi, C. Filsfils, P. Francois, and O. Bonaventure. Loop-free convergence using oFIB. In *draft-ietf-rtgwg-ordered-fib-06*, June 2012.

[20] V. Shrivastava, P. Zerfos, K. won Lee, H. Jamjoom, Y.-H. Liu, and S. Banerjee. Application-aware virtual machine migration in data centers. INFOCOM, 2011.

[21] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker. Applying NOX to the datacenter. HotNets, 2009.

[22] M. Yu, Y. Yi, J. Rexford, and M. Chiang. Rethinking virtual network embedding: substrate support for path splitting and migration. *Computer Communication Review*, 38(2):17–29, 2008.

[23] Y. Zhu and M. H. Ammar. Algorithms for assigning substrate network resources to virtual network components. INFOCOM, 2006.