Live Migration of an Entire Software-Defined Network

Dushyant Arora, Diego Perez-Botero

Abstract

In this paper, we propose an efficient mechanism for achieving *live* migration of an entire OpenFlow network. This includes OpenFlow switches, OpenFlow controllers, and any hosts associated with the network. The hosts remain online and able to communicate with one another via the network, even during the transition period. After explaining why live migration of a full Software-Defined Network (SDN) could be valuable for cloud providers and enterprises alike, we describe the considerations made and challenges that we had to overcome to implement such mechanism. We then show how our approach, which places a shim layer between the OpenFlow Controller and the OpenFlow Applications, results in successful migrations of entire SDNs while minimizing the performance hit caused by the operation.

1 Introduction

Virtualization has become commonplace in modern computing, as it enables the execution of multiple isolated instances of a software entity on top of a single physical device. This not only increases resource utilization; it also improves state encapsulation, which makes administrative tasks easier and minimizes fault propagation. While Virtual Machines (VMs) have been around from the 1960s [5], virtualized networks are a recent development. The rise of Software-Defined Networking opens a new world of possibilities. Operations that were previously limited to traditional VMs, such as live migration, can now be performed on network components. Even though live VM migration is currently used for server consolidation, load balancing, and fault tolerance, VMs are not typically isolated instances. Instead, they have tight dependencies with other VMs (e.g., as part of a multi-tier web application) and also have dependencies on the network they are running on (e.g., the network knows how to reach the VM). For this reason, it might not only be desirable to migrate a single VM, but many VMs together - moving with them the illusion of the network that they have.

1.1 Usage Scenarios

To understand why it is desirable to migrate many virtual machines together along with their associated network state, we illustrate this with examples in the context of cloud computing and that of private infrastructures.

1.1.1 Cloud Computing

- Full network migration could simplify the transition to a public cloud infrastructure. It would also enable seamless changes between cloud providers for the purpose of being able to pick another provider for pricing or feature differences. At the same time, this would allow companies to lease out their excess infrastructure in a cloud computing model.
- Services could be deployed in a Cloud Computing environment until their resource consumption stabilizes. Afterwards, the service owner might be interested in transferring the whole service (network and server configuration and state) to a private (cheaper) cloud. This is currently not an option, but our project is a step forward in facilitating such migration.

1.1.2 Private Infrastructures

- When a company acquires another company, it might be reasonable to merge infrastructures. Again, conserving service state and having little downtime could be of interest, which can be done with live migration.
- A network can be partitioned into various smaller ones to simplify administrative tasks. This way, if a live migration mechanism is available, maintenance can be performed on a subset of the infrastructure without noticeable downtime.

1.2 Software-Defined Networks and Live Migration

Live VM Migration has been studied extensively and popular hypervisors (e.g. Xen, VMware, OpenVZ) have now put reasonable solutions to practice. However, migration of network components and network state has received minimal attention (e.g., VROOM [6] demonstrated the ability to migrate a virtual router). Nevertheless, Software-Defined Networking (SDN) might change this by providing an API which gives us a way to reason about network migration without being tied to specific implementations, enabling us to structure the migration process through arbitrary steps of our choosing.

Nonetheless, Software-Defined Networking also leads to new challenges. Most notably, SDNs can run arbitrary applications, so no assumptions can be made about the controller application and protocols. In addition, network state is spread across both the controller and the network elements themselves. Last but not least, the controller applications must be provided with a consistent view of the network before, during, and after the migration.

1.3 OpenFlow

Software-Defined Networking has gained traction with the formation of the Open Network Foundation (ONF) in 2011, which has been pushing for the adoption of SDN technology called OpenFlow and is backed by industry giants such as Facebook, Google and Microsoft. OpenFlow is an open standard for programmable networks that eases the deployment of new ideas while maintaining the vendors' need for closed platforms. Without loss of generality, we will focus on the OpenFlow technology for implementation purposes. Our findings should still be easily ported onto another SDN technology.

2 Design

In order to design the Virtualized Network Migration (VNM) system, we start off by defining which state needs to be preserved and the necessary correctness guarantees. We then present an architecture for the VNM system, along with the two algorithms that will be put to test in the next section of the paper.

2.1 State

There are three main components that need to be migrated and contain state:

- **Virtual Machines (Hosts):** Live VM Migration technology is fairly mature and, as such, we can make use of it in its current form. As shown by Hines and Gopalan [3], the pre-copy algorithm used by most hypervisors leads to downtimes of less than 600 ms for different types of real application workloads. For our VNM, we will employ existing implementations of the aforementioned algorithm to migrate each VM individually.

- **Controller Application:** In SDNs, the logically centralized controller runs software that determines how the network elements are configured. This software can be anything as long as it uses the standard API to interact with the network elements, so we make no assumptions regarding what the controller application is doing.

- **Network Switches:** The switches are the components in charge of handling data traffic based on the configuration set by the controller application. As shown in Figure 1, switch state is comprised of: (1) the flow table, which contains the rules on how to handle packets; (2) the statistics, which are counters detailing historical data tied to individual flows; (3) the timers, which are used as a way to trigger the automatic removal of flow table entries after a specified time-out.

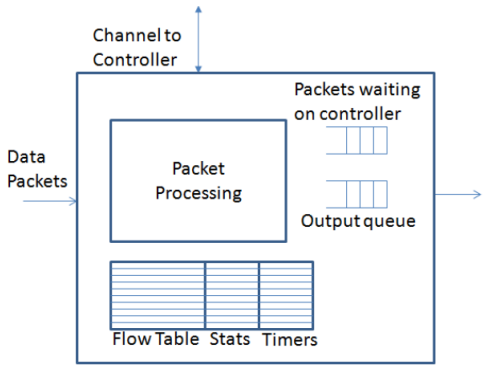


Figure 1: Switch state.

2.2 Correctness

In performing a network-wide migration, the correctness of the controller application must not be affected. By ensuring that the migration process is transparent to the controller application (i.e., the view of the network that is presented to the application is correct), we guarantee that network migration does not break the application. We make the following assumptions about the different types of network state, based on what controller applications expect according to the OpenFlow 1.1 specification [2] (an exemplar SDN):

- **Flow Table:** In order for the switches in the new topology to carry traffic, they must have the same flow rules as their counterparts in the old topology. We assume that a given command (delete or add a rule) will eventually be installed at both the old and the new switch, but do not require that it occur at exactly the same time nor require that the rules be applied in the same order unless the controller application provides explicit ordering of commands.

- **Data Packets:** Consistent with packet forwarding in general, we assume that packets can be dropped. Typically, packets would be dropped during times of congestion, whereas in the VNM system packets may be dropped due to other reasons as well. However, to the controller application and the host applications, the reason does not matter. Of course,

for performance reasons, packet loss should be minimized. Similarly, control messages between the controller application and the switches can be dropped, as they commonly traverse the same network switches; hence, falling under the same guarantees. Finally, data packets that were placed in a buffer on the switch in order to await action from the controller must not be left without action (including being dropped).

- **Timers:** We assume that timers should be preserved without a considerable deviation, but that controller applications do not rely on precise timing. Perfect timer accuracy during migration is difficult to attain. Furthermore, the OpenFlow specification does not guarantee timers to be exact.

- **Counters:** We assume that statistics are to be faithfully preserved when presented to the controller application, but do not need to be preserved in the switches. This is acceptable for now, since flow rules do not currently include the ability to use the absolute value of a counter in their decision-making process.

- **Topology:** the topology of the network itself is a form of state in relation to the overall network. Topology changes can include links failing or recovering, or whole switches leaving or recovering. In each case, the controller application is notified of the change via events. For our purpose, we treat a failure during the migration process in either network as a failure in both and notify the application.

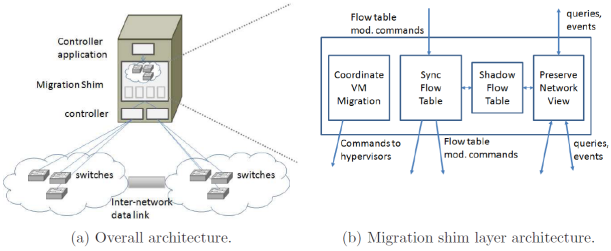
2.3 VNM System Architecture

Figure 2(a) shows the VNM System's overall architecture. Our VNM software, labeled as *Migration Shim*, runs between the controller application (e.g., pyswitch, dnsspy) and the controller run-time software (e.g., NOX, Beacon). The controller run-time software can either be executed as two instances (one per network) as shown, or as a single instance with access to the two networks. Meanwhile, the VNM shim layer has a view of both the new and the old topologies, but presents a view of a single network to the controller application. We divide the live migration into three phases:

1. **Set Up Tunnels:** Tunnels, labeled as *inter-network data link* in Figure 2(a), need to be configured between the two topologies in order to maintain communication among all the VMs (migrated and yet-to-be migrated). As will be explained in section 2.4, the use and behaviour of the tunnels differs depending on the migration algorithm being used.

2. **Migrate VMs and Synchronize Switch State:** We assume that we cannot migrate all VMs concurrently due to bandwidth limitations; while disk images will likely be shared among several VMs, the in-memory working set will be unique for each VM and can be several gigabytes in size. As will be explained in section 2.4, the actions to be taken to keep the switches in the two topologies synchronized as VMs migrate to the new network vary depending on the migration algorithm being used.

3. **Finalize:** Once all VMs are migrated, the network migration algorithm enters its final phase. This entails deleting flow rules from the old topology, deleting any tunnel-related flow rules left in the new topology, and sometimes includes copying rules from old switches to their new counterparts (depending on the algorithm being employed).



(a) Overall architecture. (b) Migration shim layer architecture.

Figure 2: VNM System Architecture.

2.4 Full OpenFlow Network Migration Algorithms

2.4.1 *Tunnel ASAP* Algorithm

The first and more simplistic migration algorithm that was conceived delegates all traffic to the old network. The rules are simple:

- Packets from migrated hosts, which originate in the new topology, jump to the old topology on their first hop.

- Packets to migrated hosts, which need to be delivered in the new topology, jump to the new topology on the last possible hop.

The main advantage of this algorithm is that *flow rules have to only be maintained in the old topology*. Nonetheless, this algorithm has two weaknesses:

1. **Communication between two migrated hosts:** let H1 and H2 be two hosts that have already been migrated to the new network. If H1 sends a packet to H2, the packet immediately jumps to the old topology. The packet then traverses the old network until it reaches the switch to which H2 was connected before migrating. Since H2 has migrated, the packet

jumps back to the new topology and gets to H2. Consequently, communication between H1 and H2 incurs in two hops through the tunnel, doubling the performance hit of the migration procedure.

2. **Final Phase:** given that the flow rules are kept up to date exclusively in the old topology, the state of the entire set of switches must be copied over to the new topology once all of the VMs have been migrated. This can generate a considerable amount of downtime and packet loss depending on the number of switches involved.

2.4.2 *Tunnel ALAP* Algorithm

The second migration algorithm is more elaborate and deals with the performance issues of the previous one. It coordinates flow rules in such a way that *packets use the infrastructure in which they originate (old or new topology) as much as possible*. While this eliminates any unnecessary jumps between topologies, it also leads to new challenges that need to be tackled. Fortunately, in terms of the final phase, downtime is minimized, given that the new topology incrementally *converges* to its required state as each host VM is migrated.

As illustrated in Figure 3, all traffic is handled in the network that the sending VM (*H_src*) is currently located in. If the packet reaches a switch that is no longer (or not yet) connected to the destination VM (*H_dst*), the packet is tunneled to the switch's counterpart in the other network to be delivered. In Figure 3, *H_dst* has previously been migrated. For this reason, a tunnel header is attached in switch *S_dst* of the old topology and *H_src*'s packet is forwarded through the tunnel path to *S_dst* of the new topology. The latter switch strips the tunnel header and delivers the packet to *H_src*. Note that, when *H_src* migrates, traffic from *H_src* to *H_dst* will flow exclusively in the new topology without any tunneling.

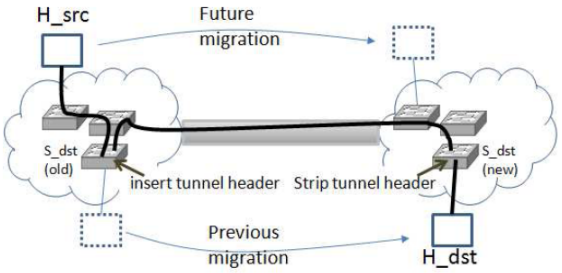


Figure 3: Example of tunneling using *Tunnel ALAP* algorithm. Old (source) topology on the left, new (target) topology on the right.

3 Evaluation

3.1 Prototype I

The first prototype's purpose was to serve as a proof-of-concept for the two migration algorithms. As such, it should provide mechanisms to test different scenarios for correctness over a variety of network topologies. Thus, an OpenFlow Network Emulator called MiniNet[1] was used, since it provides an agile framework for putting our assumptions to test. Taking into account that real hosts cannot be attached to a MiniNet network, live VM migration had to be simulated by generating traffic from different sources in a coherent way. In terms of the OpenFlow Controller, NOX[2] was picked due to its maturity and compatibility with both MiniNet and real OpenFlow switches.

Our VNM shim layer was coded as a modified NOX library. This way, NOX Controller Applications would only need to change their *import* statements to start using the VNM service. The shim layer exposes the same interface as an unmodified NOX Controller, but intercepts event callbacks (e.g. *packet_in*, *flow_removed*) in order to preserve transparency to the controller applications. Taking into account that NOX follows event-based programming and does not support multithreading, the shim layer was designed with a client-server model (Figure 4) to enable the processing of migration commands at any given time. Under this scheme, NOX applications communicate with the shim layer by importing the modified library. In turn, the shim layer communicates with the NOX Core Components, which provide the basic OpenFlow Controller functions. These components then interact with OpenFlow switches. Last but not least, the VNM Client Console coordinates the network migration process by sending commands such as *PREPARE_MIGRATION*, *MIGRATE_HOST_#* and *MIGRATE_SWITCHES* to trigger the events comprising the three different phases described in section 2.3.

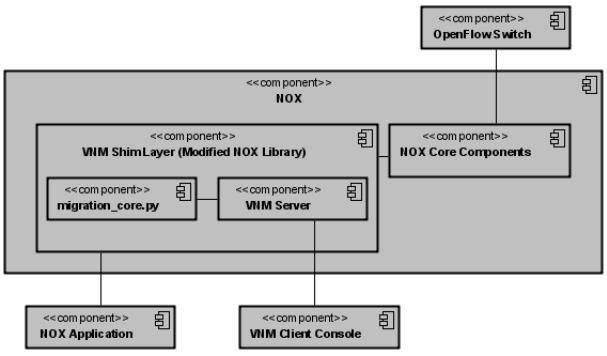


Figure 4: Component Diagram of VNM Shim Layer.

[1] http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet
[2] http://noxrepo.org/wp/

3.2 Prototype II

After debugging the two migration algorithms with the help of the first prototype, the VNM shim layer was ready to be deployed in a real (non-emulated) OpenFlow environment. The same VNM NOX library was used for this prototype, but MiniNet switches were replaced with instances of Open vSwitch[3] running on physical nodes. Meanwhile, MiniNet nodes were substituted with real VMs and Live VM Migration was carried out with the VirtualBox[4] hypervisor's Teleport service.

3.3 Test Bed

Figure 5 shows the topology that was used to evaluate the VNM system. The network to be migrated is set up on MiniNet and consists of 4 switches, 2 hosts and a NOX Controller (not shown). The NOX application being run over this topology updates the flow rules of the two switches connected to the hosts in 1-second intervals. If the traffic between the two hosts is being forwarded through the upper path, the flow rules are modified so that the lower path is now used, and vice-versa. Lastly, VLANs are used to simulate the tunneling operation.

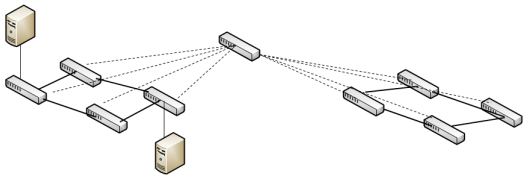


Figure 5: Test bed topology. Old network on the left, new one on the right. Tunnel switch connecting them.

3.4 Results

In order to measure the performance impact that the two algorithms have over the network, a netperf[5] TCP_STREAM test was conducted between the two hosts. From Figure 6, it can be observed that, in both cases, throughput drops from 24 Mbps to 18 Mbps when only one of the hosts has migrated, as the traffic has to be redirected through the tunnel. The main difference comes when both hosts are in the new topology and the final phase has not taken place, as discussed in Section 2.4.1. In the case of the ALAP algorithm, the throughput goes back to normal after the second host has migrated. On the other hand, the migration of the second host leads to even worse throughput (16 Mbps) until the final phase is executed.

[3]http://openvswitch.org/
[4]https://www.virtualbox.org/
[5]http://www.netperf.org/netperf/

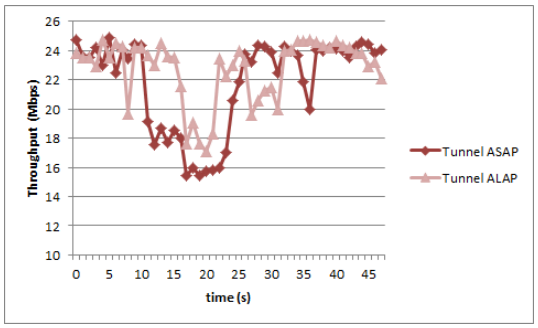


Figure 6: netperf TCP_STREAM throughput trace.

To test the migration process' impact on Quality-of-Service, Round-Trip Times (RTT) were measured with the *ping* command. From Table 1, it is clear that a noticeable increase in RTT presents itself, mainly because of the additional hops associated with the tunneling. Nonetheless, RTTs of roughly 0.4 ms, as the ones detected during migration, are more than acceptable for most (if not all) networked services.

Table 1: Ping Results Before and After Migration.

	Algorithm 1		Algorithm 2	
Measurement	Before	During	Before	During
Avg. RTT (ms)	0.247	0.434	0.231	0.441
RTT s.dev. (ms)	0.516	0.867	0.516	0.949

Our last test used the D-ITG platform [1]. We set up a DNS server on one host and a DNS client on the other one. A 25-second performance test was run for each algorithm, during which the three migration phases described in Section 2.3 were performed. Roughly 25,000 packets were generated in the 25-second time lapse, with 0.44% packet loss reported for the ASAP algorithm and 0.39% packet loss reported for the ALAP algorithm. The higher packet loss value for ASAP can be explained by the final phase problem discussed in Section 2.4.1.

4 Related Work

Up until now, live migration of network components has focused on fine granularity. While the VROOM project [6] takes advantage of control and data plane separation to perform live migration of a virtual router, Pisa et al [4] propose a strategy to migrate an OpenFlow switch to another physical host inside the same network. Our proposed project goes for a coarser approach, taking an OpenFlow network and its associated hosts as a single entity to be migrated consistently with imperceptible downtime.

5 Conclusions

Software-Defined Networks provide powerful abstractions that enable us to perform administrative tasks that were previously limited to other areas of computing. Live migration is a powerful tool that can be of great value in Cloud Computing and private enterprise contexts alike. In this paper, we examined two algorithms for performing live migration of entire OpenFlow networks. Our results indicate that our proposed solutions are feasible in real SDN deployments, with acceptable throughput degradation and little packet loss. This is especially true to the ALAP algorithm. Last but not least, the VNM Shim Layer architecture proves to be effective and makes it possible for existing OpenFlow controller applications to start using live migration services without considerable changes in their source code.

References

[1] A. Botta, A. Dainotti, and A. Pescape. Multi-protocol and multi-platform traffic generation and measurement. *INFOCOM 2007 DEMO Session*, May 2007.

[2] O. N. Foundation. Openflow switch specification 1.1.0. http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf.

[3] M. R. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. *Proceedings of the 2009 ACM SIGPLANSIGOPS international conference on Virtual execution environments VEE 09*, page 51, 2009. URL http://portal.acm.org/citation.cfm?doid=1508293.1508301.

[4] P. S. Pisa, N. C. Fernandes, H. E. T. Carvalho, M. D. D. Moreira, M. E. M. Campista, L. H. M. K. Costa, and O. C. M. B. Duarte. Openflow and xen-based virtual network migration. In A. Pont, G. Pujolle, and S. V. Raghavan, editors, *WCITD/NF*, volume 327 of *IFIP International Federation for Information Processing*, pages 170–181. Springer, 2010. ISBN 978-3-642-15475-1. URL http://dblp.uni-trier.de/db/conf/ifip6/wcitd2010.html#PisaFCMCCD10.

[5] VMWare. Virtualization basics. http://www.vmware.com/virtualization/history.html.

[6] Y. Wang, E. Keller, B. Biskeborn, J. van der Merwe, and J. Rexford. Virtual routers on the move: live router migration as a network-management primitive. *SIGCOMM Comput. Commun. Rev.*, 38:231–242, August 2008. ISSN 0146-4833. doi: http://doi.acm.org/10.1145/1402946.1402985. URL http://doi.acm.org/10.1145/1402946.1402985.