Efficiently Migrating Stateful Middleboxes

Vladimir Olteanu
Universitatea Politehnica Bucuresti
vladimir.olteanu@cs.pub.ro

Costin Raiciu
Universitatea Politehnica Bucuresti
costin.raiciu@cs.pub.ro

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*packet-switching networks*

Keywords

Middlebox, Migration

1. INTRODUCTION

Middleboxes that hold per-flow state and perform Layer 4+ processing are widely deployed in the Internet today: a recent study shows their presence on at least a third of the studied paths [5]. These middleboxes provide functionality ranging from security to performance optimization, and are becoming ubiquitous with time.

To reduce costs and enable fast functionality updates there is an ongoing trend of migrating away from specialized hardware implementations of middleboxes to software running on commodity servers [9].

Programmable switches (such as OpenFlow) coupled with x86 machines have been proposed as the natural architecture to create scalable middleboxes that are also easy to deploy and update [3]. The basic recipe is very simple: a collection of x86 servers are connected to an OpenFlow switch, which is in turn "on-path" for the traffic. The servers implement distributedly the functionality of a single middlebox (possibly in virtual machines), such as carrier-grade NAT or firewall. The programmable switch is a key ingredient, splitting load between the machines.

To make such distributed middleboxes scalable, we need ways to seamlessly move processing and its associated flow state between local or remote servers. This would allow the platform to deal with load surges by adding servers, and to efficiently scale down by shutting machines off. Processing could even be migrated to different middleboxes in other parts of the world to optimize other aspects such as user-perceived delay.

2. EXISTING APPROACHES

Flow processing middleboxes keep most of their state in memory to reduce packet-level delays. Hence, migrating flow processing boils down to efficient memory migration. To this end, process migration algorithms [7] are the obvious starting point.

SIGCOMM'12, August 13–17, 2012, Helsinki, Finland.
ACM 978-1-4503-1419-0/12/08.

There are three basic techniques used for process migration: *Pre-copy:* Processing is performed by the original host and the memory pages are copied to the destination machine, in succesive rounds. The first round copies all active pages, while subsequent rounds only copy "dirty" pages. *Stop-and-copy:* The source machine stops processing; the memory pages are copied and processing resumes on the destination machine. *Demand migration:* The destination machine starts processing and faults unavailable pages across as they are accessed.

Process migration algorithms typically use two of the three techniques above and work reasonably well because of the principle of locality: if a page is accessed, it is likely that it will be accessed again soon. That is why the combination of a few pre-copy rounds followed by a quick stop-and-copy phase also works well for virtual machine migration [2].

Our focus is, however, on network processing that keeps per-flow state. Say we're using the same algorithm to migrate flow-states. Here is what will happen between two successive pre-copy iterations: a significant number of states will die off, because most flows are short-lived [8], a significant number of new states will be established and a seemingly random subset of the existing states will be updated, depending on which packets hit the middlebox in the meantime.

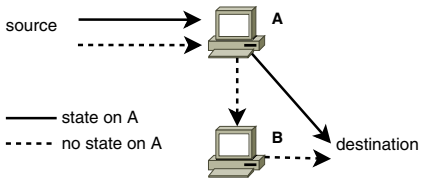
The principle of locality does not apply to flow-states and this makes pre-copying ineffective. Migration causes a long period of downtime due to the stop-and-copy phase which has to copy all the state across.

3. STATE MIGRATION

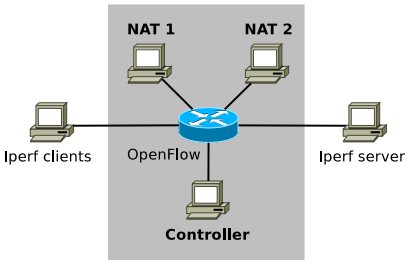
Stateful network processing allocates state for each flow (e.g. 5 tuple), and memory accesses are made to this flow-state when packets belonging to the flow arrive at the middlebox. Such processing also maintains global state (e.g. state about flow aggregates), but this is typically much smaller in size.

Our focus is on efficiently migrating per-flow state. The basic idea is to allow short flows to die and their corresponding state to be deleted before any flow-state transfer takes place. To achieve this goal, we run in parrallel both the source and the destination processing for a short period of time. When migration begins, a small set of pages is copied to B containing code and invariant global state. After this, the algorithm consists of two overlapping phases: the idle phase and the iterative freeze-and-copy phase.

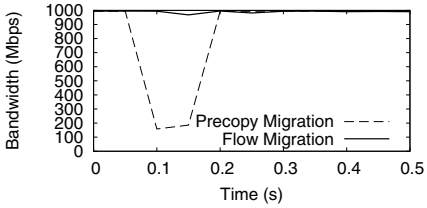
In the *idle* phase, all traffic is initially forwarded to A. A stops establishing new state for flows it hasn't seen yet. All



(a) Daisy Chaining



(b) Experimental Setup



(c) Impact of migration on bandwidth

packets that hit A but don't have any matching state are forwarded to B, who processes them normally.

The *idle* phase takes advantage of the fact that most flows are short-lived. Rather than transfer those states, they can be kept on the original machine until they expire while, at the same time, allowing states for new flows to be established on the other machine. The two machines are daisy-chained (packets will always hit A first, regardless of direction; see figure 1(a)).

The *freeze-and-copy* phase is iterative and only starts when the rate at which states expire on A falls below a certain threshold; the remaining states are likely long-lived and need to be transfered. A freezes some of its remaining states and begins transferring them to B. All packets that would modify a frozen state are buffered by A. When the transfer finishes, all frozen states are deleted and the buffered packets are forwarded to B. The process is repeated until all states are copied to B. This phase only deals with long-lived flows and freezing aims to reduce the number of dropped packets. Daisy-chaining continues until the *freeze-and-copy* state finishes. At this point, there is no flow state left on machine A and traffic is redirected in the OF switch to hit machine B directly.

4. APPLICATION: CARRIER-GRADE NAT

We have implemented in Click [6] a scalable carrier-grade NAT with reactive load-balancing as a proof of concept for our flow state migration algorithm. Current best practices [4] along with OpenFlow's limitations [1] have driven our design of the NAT. Internal (IP, port) tuples are dynamically mapped onto their external counterparts; all connections corresponding to one such mapping constitute a flow whose state can be independently frozen and moved. The smallest migration goal is made up of all flows that share a certain external IP.

We have run experiments with a Gigabit Ethernet link saturated by 100 running instances of iperf, using the experimental setup shown in Fig.1(b). NAT box 1 is initially active, and it also has state for 65,000 fictive connections, emulating an external IP with the port space fully used. Each state is 24 bytes in size, bringing the total to roughly 1.5 MB. We migrate the flow states to NAT 2 both via our flow migration algorithm and via a single stop-and-copy round. Figure 1(c) shows the impact of the migration process on the bandwidth. While freeze-and-copy incurred only some minor disruption due to minor RTT inflation caused by the freeze time, the stop-and-copy round caused packet loss disrupting the iperf flows.

The downtime for stop-and-copy migration increases linearly with the number of flows and the size of the flow state. The NAT per flow state is small, but migrating more connections will bring proportionally more downtime which is unacceptable for production use.

5. NEXT STEPS

An open question is how effective precopy is with real traffic traces. Our simulations[1] with zipf-like flows sizes show that precopy has limited effect, however we need to experiment with real traffic traces to get a full picture.

Ideally, migration should be provided by the OS without changing the apps, and this is our end goal. It would allow us to migrate existing applications such as Bro or Snort without rewriting them, and giving our techniques wider applicability.

Achieving this goal is challenging. Running both source and destination processing in parallel is tricky as it requires efficient synchronization of accesses to global state. Done inneficiently, such synchronization can undo the performance benefits of flow-migration. Further, to implement daisy chaining, the OS must know which flows have state associated and which not; acquiring this knowledge without application support seems feasible only if both flow definitions and state expiration mechanisms are known to the OS.

6. REFERENCES

[1] Openflow switch specification, version 1.0.0.
[2] C. Clark and et al. Live migration of virtual machines. In *Proc. NSDI*, 2005.
[3] A. Greenhalgh and et al. Flow processing and the rise of commodity network hardware. *SIGCOMM CCR*, 39(2):20–26, 2009.
[4] S. Guha, K. Biswas, B. Ford, S. Sivakumar, and P. Srisuresh. NAT Behavioral Requirements for TCP. RFC 5382, Oct. 2008.
[5] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend tcp? In *ACM IMC*, 2011.
[6] E. Kohler, R. Morris, B. Chen, J. Jahnotti, and M. F. Kasshoek. The click modular router. *ACM Transaction on Computer Systems*, 18(3):263–297, 2000.
[7] R. Lawrence. A survey of process migration mechanisms.
[8] L. Peterson. Inter-as traffic patterns and their implications. In *Proc. IEEE GLOBECOM*, 1999.
[9] V. Sekar and et al. The design and implementation of a consolidated middlebox architecture. In *Proc. NSDI*, 2012.

[1]Not shown here; these results led us to only consider stop-and-copy as the alternative migration solution