A Prototype Implementation of MOOSE on a NetFPGA/OpenFlow/NOX Stack

Daniel Wagner-Hall
University of Cambridge Computer Laboratory
dwh@cantab.net

ABSTRACT

Ethernet does not scale well to large networks. The flat MAC address space, whilst having obvious benefits for the user and administrator, is the primary cause of this poor scalability. MOOSE — Multi-level Origin-Organised Scalable Ethernet — has been presented as a viable solution to this problem. In this paper, we present an evaluation of MOOSE through a prototype switch, built using OpenFlow and NOX on the NetFPGA network prototyping platform. We find MOOSE to be a worthwhile protocol, achieving its proposed improvements, and worthy of future research.

1. INTRODUCTION

Ethernet, first devised in the '70s [7], has evolved to match the changing landscape of network deployments over the decades. Through preserving its self-same addressing and frame-structure for backward compatibility, it has reached a hurdle which it cannot overcome. Multi-level Origin-Organised Scalable Ethernet (MOOSE) [11] has been proposed as a solution to the problems outlined in Section 1.1. This paper describes the development and evaluation of a realistic prototype MOOSE switch using OpenFlow [6] on the NetFPGA platform [8], providing an evaluation of MOOSE, as well as the development platforms.

1.1 Scalable Ethernet

Ethernet networks were originally small (tens to hundreds of hosts) shared-medium networks. The important property of Ethernet MAC addresses was that they were universally unique, and that this uniqueness could be easily provided. The format of MAC addresses was thus defined to be six bytes: three bytes of organisationally unique identifier (OUI) allocated to the device's manufacturer by the IEEE, followed by three bytes allocated by the manufacturer. As networks became larger and switched networks became normal, these addresses were preserved for backward compatibility and interoperability. While this flat address space requires no configuration of hosts or switches (a globally unique MAC address can be used anywhere), this leaves each switch with the task of learning and storing the location of every addressable device on its network. Ethernet devices frequently use broadcast (e.g. ARP queries), and so every address is seen by every switch on a network, and every address is stored with its physical location (port). This forwarding table must be stored in fast memory, as it is checked for every packet received. Typically, content-addressable memory (CAM) is used for its speed, particularly as 10Gb/s Ethernet becomes ubiquitous, and as the 40Gb/s and 100Gb/s Ethernet standards have just been ratified.

Growth of networks necessitates growth of this forwarding table; however, increasing the capacity of CAM without sacrificing speed, and whilst constraining energy consumption is hard [10]. In modern switches, this places a capacity constraint of the order of 16,000 entries [1]. Virtual hosts are included in this table, magnifying the problem. Though higher capacity forwarding databases exist, they are constrained to very high-end switches. On moderately large networks, full databases are a significant problem — if databases become full, entries are discarded, and frames for unknown addresses are forwarded to all ports, resulting in heavy congestion, especially a problem for low-capacity edge links. Ethernet's flat address space allows for no natural routing protocol to be used other than learning locations of addresses as they send frames, as addresses cannot be aggregated together as takes place in higher-level Internet Protocol routing.

As well as the size of the forwarding database, Ethernet's inability to handle networks containing loops gives another constraint to scalability. The Rapid Spanning Tree Protocol, RSTP [3, §17], must remove loops by disabling redundant links. In a dense network, for instance in data-centres, RSTP will disable a large proportion of links, constraining frames to suboptimal routes and introducing bottlenecks where redundancy may have been planned.

1.2 MOOSE

MOOSE mitigates these issues by introducing routing information into MAC addresses through adding hierarchy, whilst preserving backward compatibility and interoperability. MOOSE operates by dynamically assigning new hierarchical MAC addresses (within the Locally Administered Address space) to each host on the network. This dynamically-assigned address is referred to as a MOOSE address to avoid confusion with static, manufacturer-assigned MAC addresses, but is itself a valid MAC address guaranteed not to collide with any manufacturer-assigned address. Every frame entering the network has its source address rewritten in-place to the sending host's MOOSE address by the first MOOSE-aware switch which it traverses. The destination address is left intact, in the expectation that it is already a MOOSE address — hosts' ARP caches will only contain MOOSE addresses, as some MOOSE-aware switch will have already performed relevant address rewriting for any received frames, and a host's manufacturer-assigned MAC address is never seen beyond that host's nearest MOOSE switch. The nearest MOOSE switch to a host rewrites any frames destined for that host's MOOSE address with that

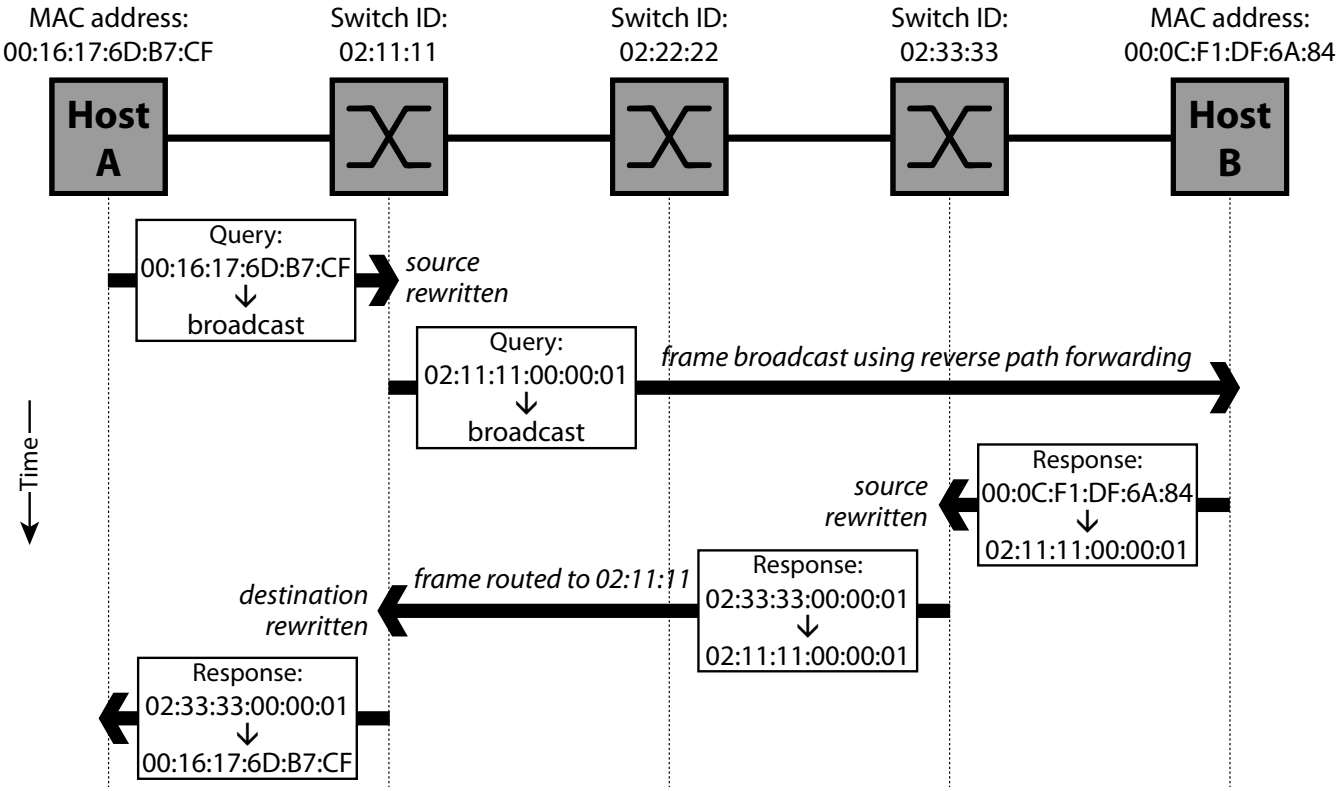


Figure 1: Sequence diagram of a request and response using MOOSE.

host's manufacturer-assigned MAC address, so that the host knows to receive them, and is not required to be aware of MOOSE's presence. In such a way, no modification to hosts is required for MOOSE to be used in a network.

To facilitate this routing information, MOOSE introduces hierarchy in addresses. Every MOOSE switch has an identifier of between one and four bytes long. A MOOSE switch allocates an identifier to each host for which it performs address rewriting which is of length $6-|\text{switch identifier}|$ bytes. A host's MOOSE address is formed by concatenating the switch identifier with the host identifier, forming a complete 6-byte address. Switch identifiers have their length encoded into them, and so are easily extractable from a MOOSE address.

In such a way, switches only need to have entries in their forwarding table per-switch rather than per-host (as well as entries for locally attached hosts), as a switch must only forward a frame to the host's nearest MOOSE switch, and that switch will forward the frame to the host. As well as reducing the size of the forwarding table, this scheme allows a shortest-path routing protocol such as OSPF [4] to operate between switches. Using such a routing protocol, the Rapid Spanning Tree Protocol is also no longer required — not only is shortest path routing possible, but also no links require disabling. Both of the major scalability issues of Ethernet are accordingly addressed.

MOOSE brings some other benefits in addition to those listed above, such as reduction of broadcast traffic. All of these improvements are described in more detail by Scott *et al.* [11].

2. PREPARATION

As no realistic prototype MOOSE switch existed, one was developed to evaluate MOOSE's claimed improvements and practical constraints.

2.1 NetFPGA and OpenFlow

The aims of the prototype were to evaluate MOOSE's claims to reduce forwarding table size from $O(\text{hosts})$ to $O(\text{switches})$, investigate the practicality of using routing protocols in layer 2, and to evaluate inter-operation with legacy Ethernet networks. NetFPGA was an obvious choice of platform to use to this end, as it allows rapid prototyping of line-rate operation of switches.

The use of OpenFlow allowed even more rapid prototyping and flexibility; for example, implementations of different routing protocols could be more easily injected for direct comparison. OpenFlow also expands the hardware available for testing — though we currently have no results using proprietary switches, we have plans to, as NetFPGA's limited four ports serve as a significant restriction to setting up test networks. Writing the switching logic in C++, rather than Verilog, also gives sample code with a wider reach of comprehension among the research community, and using OpenFlow rather than raw Verilog also allows more researchers to try the sample code without requiring specialist NetFPGA cards (for instance on PCs with multiple network interfaces).

3. IMPLEMENTATION

NOX is a library available in C++ and Python giving an abstract, event-driven interface to OpenFlow. An OpenFlow MOOSE switch was written on top of NOX [2] in Python,

aiming to give a simple, high-level implementation. Basic testing showed that the simplest of OpenFlow switches written using NOX in Python (i.e. an Ethernet learning switch) performed an order of magnitude slower than their C++ equivalents (100Mb/s vs 1Gb/s). Though the aims of this prototype were to measure behaviour, rather than speed, it was felt that this significant slow-down was worth avoiding, so the switch was re-written in C++.[1]

Modularity

Some of the principal tenets of the NetFPGA project are modularity and reusability. These were upheld in the MOOSE switch. As an example, the OSPF Internet routing protocol was adapted for use with MOOSE switch identifiers by stripping PWOSPF [12] to its minimum required functionality, and implemented as a routing module. This module can be trivially swapped with implementations of other existing or novel routing protocols such as IS-IS [5] to compare their behaviour.

Testing

An extensive regression test suite of automated unit tests was performed on the C++ switching-logic code as it was written. Manual integration tests were performed by connecting computers to one or more NetFPGAs, sending well-defined traffic through the network and verifying that the correct behaviour was noted (frames received, addresses rewritten, etc.).

4. EVALUATION

Several experiments were performed to compare the behaviour of MOOSE with that of Ethernet.

4.1 MOOSE

12 NetFPGA cards running the stock OpenFlow switch loaded with the MOOSE switching logic, and 24 standard 1Gb/s network interfaces controlled by Linux computers were used. Before all experiments, the ARP cache of each host was cleared, and the OpenFlow switches reset to clear their forwarding tables.

Forwarding Table Size

In the common case, MOOSE offers to reduce the number of entries in switches' forwarding tables, allowing more entries in the same memory. MOOSE switches are claimed by Scott *et al.* [11, §IV] to require O(switches) entries in a switch's forwarding table (with a small number of entries for locally attached hosts), compared to Ethernet's O(hosts) entries. For dense networks (with more hosts than switches, very much the common case), this offers a significant reduction in the number of entries in the forwarding table. For sparse networks (seldom seen), however, this may increase the size of the forwarding table. Two experiments were performed, illustrating both the dense and sparse network cases. Exactly MOOSE's proposed forwarding table sizes were seen.

As NetFPGAs have only four Ethernet ports, particularly dense networks could not be used for testing, but hosts could outnumber switches 2:1, still showing improvement. The use of OpenFlow allows more dense networks to be used in the future, using proprietary OpenFlow-enabled switches. 12 switches were attached in a line, with two hosts connected to each switch, and each host was made to communicate with each other host. In this way, every host received frames from each other host, and every switch processed frames from each other switch. As predicted, each switch's forwarding table contained one entry per foreign switch, and one entry per locally attached host, giving 13 entries in total. When Ethernet (without RSTP) switching logic—rather than MOOSE switching logic—was used, each forwarding table contained one entry per host, giving 24 entries in total — many more than MOOSE's 13.

In the MOOSE network, there was a short delay after turning the switches on before the hosts could communicate with each other, while the routing protocol converged. On the Ethernet network, no such delay was encountered. A similar delay could be envisaged if a spanning tree protocol were being used.

To test the sparse network case, the same dozen switches were left in a line, but with two hosts attached to one switch, and none to the others. The two hosts communicated with each other. In the MOOSE network, all switches' forwarding tables contained all of the other switches (12 entries), and the switch with hosts attached also contained an entry for each of those hosts, totalling 14 entries. In the Ethernet network, all switches' forwarding tables contained one of the hosts (as its frame's destination was previously unseen, so the frame was broadcast), and the switch with attached hosts also had the other host in its table (2 entries in total). MOOSE gave a significant increase in forwarding table size, though this sparseness of network is expected to be incredibly rare in situations where forwarding table size is a problem.

Significant multicast traffic was noted across the network due to the routing protocol being used (OSPF is known to be multicast-heavy), adding some congestion which MOOSE strives in general to avoid. A more multicast-light protocol (such as IS-IS) is worth considering, but evaluation of alternatives is left for future experimentation.

Shortest Path Routing

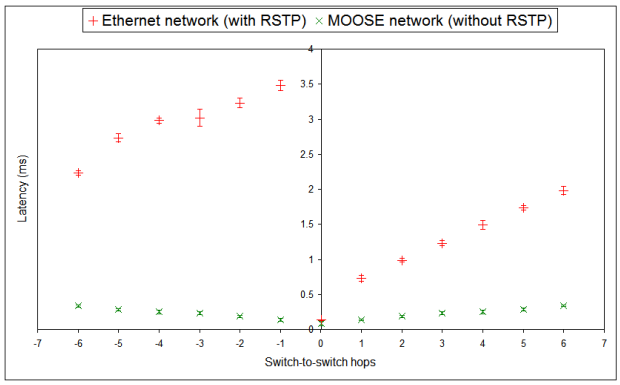
Two pathological cases exist for Ethernet, both involving loops. Should no spanning tree protocol be used, any broadcast traffic on the loop will be continually re-forwarded around the loop until the loop is broken. In a test with three switches in a loop and a host attached to each, some 120,000 copies of a broadcast packet (sent once by its source) were observed around the loop after 10 seconds. In the same loop with MOOSE switches, the frame was received exactly once by each host, and no such broadcast storm occurred.

The other, common, pathological case is that of RSTP disabling links. 12 MOOSE switches were set up in a loop, with a single host connected to each switch. One host pinged each other host[2]. The times taken for these pings were recorded. The experiment was then repeated using Cisco Catalyst 3500-48 100Mb/s Ethernet switches with RSTP enabled. Unfortunately 1Gb/s switches were not available, so the times were not directly comparable, but the effects are still clear.

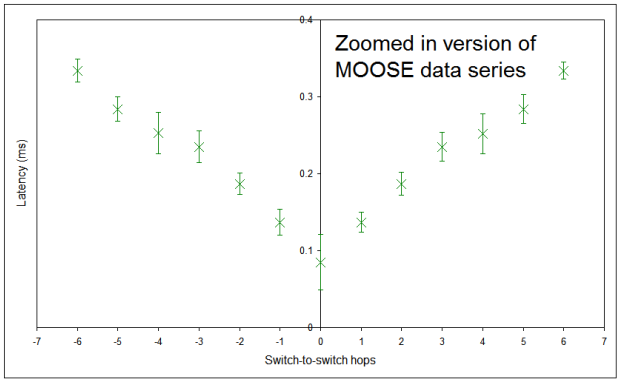
In the Ethernet network, RSTP disabled a single link to remove the loop from the topology, so any traffic whose shortest path to its destination included that link was forced

[1]The source code is available online at {TODO: Fill in stable URL} for inspection and use.

[2]Specifically, the command `ping -c 10000 -s 1400 -f <destination IP>` was used



(a) Ethernet network (with RSTP) and MOOSE network (without RSTP)



(b) Enlarged version of just MOOSE network (without RSTP)

Figure 2: Graphs of ping times around a loop in both Ethernet network with RSTP and MOOSE network without RSTP (above), and enlarged version of graph with just MOOSE network (below). Error bars indicate ±1 standard deviation.

to use a suboptimal route, going the other way around the loop. In the worst case, traffic between the hosts whose adjacency was disabled would need to cross all twelve switches, despite the shortest path being simply across two. In the MOOSE network, because no links were disabled (as shortest-path routing could be used, rather than RSTP), the optimal path was always taken.

Figure 2 shows these ping times. The absolute value of the switch-to-switch hops measure is the number of links between switches on the shortest path between the source and destination, and the sign indicates the direction in the loop (specifically, the negative values included the disabled link, where the positive values did not). It is clear that the Ethernet network (with RSTP) forced distinctly suboptimal paths to be used, where in the MOOSE network, optimal paths were always used.

Mobility

To test that a simple care-of-forwarding mobility scheme outlined in [11, §IV-E] functions, a long TCP session typical of a large file transfer was initiated between two hosts attached to different switches. One of the hosts was physically disconnected from its switch, and reconnected to another switch on the network. Within approximately three seconds of physical disconnection from the original switch (including approximately two seconds of physical disconnection time), the TCP session had resumed. Mobility was shown to work flawlessly.

4.2 NetFPGA, OpenFlow and NOX

OpenFlow provided an excellent platform for prototyping, enabling the rapid and flexible development it claims to offer. It was, however, noted that OpenFlow rules had to be created describing actions on frames between every pair of hosts, rather than one per host or switch, because a single lookup of all frame headers is done per frame, whereas *both* source and destination Ethernet address are used for switching in MOOSE, leading to many more rules being created (and needing to be managed) than are strictly necessarily. The current multiple-tables work being done on OpenFlow [9] may alleviate this issue.

The delay introduced into the switching path by using OpenFlow (and so having a Linux PC enter the switching path, and introducing DMA packet copying to provide that PC with packets) meant that time-measurements could not be used to compare MOOSE's likely increased latency to that of Ethernet, and a Verilog NetFPGA or similar MOOSE switch implementation will be required for those measurements.

NOX provided a very useful abstraction over OpenFlow, but was found to be a somewhat immature project, due specifically to the lack of convenience-functions around the rule-management classes (e.g. `ofp_flow_mod`), and lack of provision of sufficient modelling of the state of rules in the switch. To modify existing rules, one must know which rules exist in the switch, but this data cannot be retrieved from the switch. A general-purpose reusable class for storing these rules in memory could be useful, but is absent.

NetFPGA was found to be a useful prototyping platform, though its full benefits were not exploited in these experiments. The limitation of four Ethernet ports restricted the scale of experimentation which could be done, but the use of OpenFlow mitigated this.

5. CONCLUSIONS AND FUTURE WORK

MOOSE was found to provide the proposed improvements in terms of forwarding table size, optimal routing, and mobility. OpenFlow provided a useful, flexible platform for developing the prototype MOOSE switch.

This work did not address the ELK [11, §IV-D] extensions to MOOSE to address reduction of broadcast traffic, and improvements to DHCP and ARP. ELK should be further investigated and evaluated.

As this proof-of-concept prototype proved successful, a future prototype should be created to analyse low-level timing differences between MOOSE and Ethernet. The existing prototype, however, continues to be useful, for example to experiment with different routing protocols. NetFPGA provided a useful and extensible platform for current and future development of prototypes, and is likely to be used natively for the next prototype.

6. ACKNOWLEDGEMENTS

I am grateful for much advice from Andrew Moore, Malcolm Scott, Jon Crowcroft, David Miller and Emma Win-

ston.

7. REFERENCES

[1] 3Com Corporation. Switch 5500G 10/100/1000 family data sheet. Online, http://www.3com.com/other/pdfs/products/en_US/400908.pdf. Retrieved 2010-03-14.

[2] 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):105–110, 2008.

[3] IEEE. 802.1D Standard for Local and metropolitan area networks: Media Access Control (MAC) Bridges, June 2004.

[4] IETF. RFC2328 OSPF Version 2, April 1998.

[5] ISO. ISO/IEC 10589:2002(E) Telecommunications and information exchange between systems — Intermediate System to Intermediate System intra-domain routeing information exchange protocol for use in conjunction with the protocol for providing the connectionless-mode network service (ISO 8473), November 2002.

[6] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Computer Communication Review*, 38(2):69–74, April 2008.

[7] R. M. Metcalfe and D. R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[8] J. Naous, G. Gibb, S. Bolouki, and N. McKeown. NetFPGA: Reusable router architecture for experimental research. In *PRESTO '08: Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, pages 1–7, New York, NY, USA, August 2008.

[9] OpenFlow Team. Discussion of multiple tables in OpenFlow. Available from http://www.openflowswitch.org/wk/index.php/OpenFlow_Meeting_Notes_4-20-2010 — Retrieved 2010-05-01.

[10] K. Pagiamtzis and A. Sheikholeslami. Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey. *Solid-State Circuits, IEEE Journal of*, 41(3):712 – 727, March 2006.

[11] M. Scott, A. Moore, and J. Crowcroft. Addressing the scalability of Ethernet with MOOSE. In *ITC 21 First Workshop on Data Center – Converged and Virtual Ethernet Switching (DC CAVES)*, Sept. 2009.

[12] Stanford. Pee-Wee OSPF (PWOSPF) Protocol Details. Available from http://yuba.stanford.edu/cs344/pwospf/ — Retrieved 2010-03-25.