Mininet Clustering

Aaron Blankstein, Scott Erickson, and Marcela Melara
Dept. of Computer Science
Princeton University
{ablankst, scottme, melara}@cs.princeton.edu

ABSTRACT

This paper presents a distributed version of Mininet, an Openflow network emulator. Normally, Mininet emulates an entire network of hosts and switches, each running as a seperate process within a Linux container, all on the same machine. Unfortunately, emulating networks that require more resources than a single machine are currently impossible in Mininet. By extending Mininet with cross-machine virtual links, this work allows researchers to emulate larger and more complex networks. Further, we demonstrate that network topologies can be automatically partitioned across machines, alleviating the otherwise burdensome task of doing so by hand.

Keywords

Emulation, graph cutting, GRE tunneling, Mininet, rapid prototyping

1. INTRODUCTION

One problem network researchers often face is how to test new network topologies and protocols in a quick, inexpensive, and realistic manner. As many researchers can attest, building testbeds costs valuable time and money, with the added issue that testbeds are limited in size and thereby do not represent realistic networks. On the other hand, while simulators are a common alternative to testbeds, they lack realism for two reasons: the simulation code is not the same as the code that would be deployed in a real network, and they are not interactive. Mininet [7] was developed to solve these issues. Mininet is a network emulator which runs on a single computer and leverages Linux features such as virtual Ethernet pairs and separate network namespaces to emulate many-host networks.

However, Mininet has a significant limitation due to only running on one computer: a single computer can only emulate a fixed number of network entities. What network researchers really want to emulate, though, are data center sized network, probably containing thousands of virtual hosts. Further, even if researchers have access to many machines, they are unable to use them to emulate more hosts because Mininet currently cannot hook them up. To resolve this issue, we have extended Mininet to support execution in a cluster. What this means is that Mininet now has the ability to split network topologies among two or more machines while maintaining its original emulation functionalities. The distributed hosts and switches tunnel their communication using Generic Routing Encapsulation (GRE). All the while, the emulated network is agnostic to this distribution.

The rest of this paper is organized as follows: we describe the design changes we made to Mininet to create Mininet Clustering in Section 2, followed by the implementation details in Section 3. Section 4 is dedicated to evaluating Mininet Clustering's performance. In Section 5, related work is discussed. We conclude and touch on future work in Section 6.

2. DESIGN OF MININET CLUSTERING

In order to add the capability to support clustering, we needed to make two major changes to the original Mininet design:

1. Add support for GRE tunneling.
2. Add functionality for automatically splitting a network topology across machines in the cluster.

To best illustrate what changed in Mininet Clustering, it is useful to contrast our design decisions with the original Mininet design.

2.1 GRE Tunneling

Since Mininet was originally designed to only be run on a single laptop or desktop computer, all of its features exploit local features, more specifically processes, and virtual Ethernet pairs in network namespaces in Linux. Mininet Clustering still leverages these Linux features on each laptop to facilitate portability from Mininet to our work.

A Mininet network has four components: Links, Hosts, Switches, and Controllers. Each link between a node is a virtual Ethernet (veth) pair which acts like a wire connecting two virtual interfaces. These appear as fully functional Ethernet ports to all system and application software. The hosts in Mininet are shell processes which are each located within their own network namespace, so that each host can have its own set of virtual Ethernet interfaces, as well as a pipe to a parent Mininet process. Mininet uses OpenFlow [8] switches and controllers to best emulate the packet delivery semantics of hardware switches. Note that Mininet already supports remote controllers with the requirement that the machine running the switches has IP-level connectivity to the controller. Figure 1 shows how the various components of a simple network with two virtual hosts network are laid out and interact with each other on a single machine.

In Mininet Clustering, we adopted the host, switch, and controller design of the original Mininet. Each of our switches is an Open vSwitch [10] and we use OpenFlow controllers

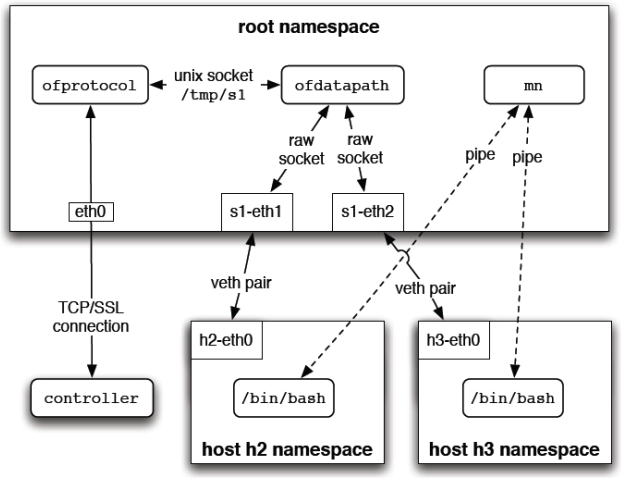

Figure 1: The components and their connections in a simple two-host Mininet network.

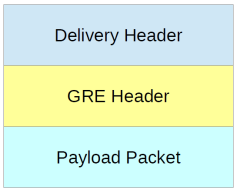

Figure 2: The GRE protocol encapsulates the payload packet both with a GRE header and then the header of the delivery protocol.

as well. However, the link design was not sufficient for our purposes. Thus, to allow for communication over a physical network connecting both machines, Mininet Clustering requires an additional type of link which will support this functionality; we chose GRE tunneling.

The Generic Routing Encapsulation (GRE) protocol is broadly speaking a protocol for performing encapsulation of an arbitrary network layer protocol over another arbitrary network layer protocol [3]. Thus, when a system has a packet that needs to be encapsulated and routed, a two-step process occurs. First, the packet to be sent, the "payload packet", is encapsulated in a GRE packet. In the second step of the encapsulation process, the GRE packet is then encapsulated in another protocol, the "delivery protocol", which then manages the actual forwarding of the packet. Figure 2 is a visual of an encapsulated payload packet. A major advantage of this protocol is that delivery layer packets and delivery layer-encapsulated GRE packets will look the same when forwarded at the delivery layer.

We use GRE to create a virtual ethernet pair across across two machines with the `iproute` tool:

```
ip link add [interfaceName] type gretap
    remote [remoteIP] local [localIP] key
    [number]
```

In this case, the GRE protocol is running over IP (the delivery protocol) to deliver ethernet packets (the payload).

Each machine in the cluster runs one instance of Mininet

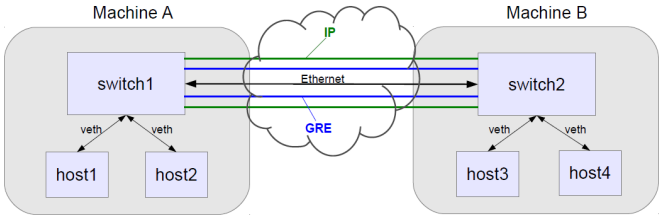

Figure 3: How GRE tunneling is implemented in Mininet Clustering in the example of a simple two-switch topology.

Clustering. Each instance will determine how many GRE links are required and what machines those links should point to. Once the topology is distributed, Mininet Clustering will assign a name for each GRE interface. If more than one GRE link exists for a given pair of machines (i.e., machine A and B have two virtual links between them), the `key` value is used to create multiple GRE links which can be demultiplexed by the OS.

An example with a simple topology distributed on two machines will best illustrate GRE tunneling in Mininet Clustering. Take a simple two-switch topology running on two machines as in Figure 3, each switch having two hosts; each half of the topology resides on a separate machine each running an instance of Mininet Clustering. To simulate the connection between the two switches, the original Mininet would just create a veth pair between the two nodes. This, however, does not allow links across two machines.

Instead, Mininet Clustering creates a GRE link between the two machines encapsulating an Ethernet connection in an IP connection, while the virtual hosts still treat the link between the two switches as a normal ethernet pair.

For this small topology, the `key` value can be omitted since there is only one single connection between the two machines. However, this value is useful for more complex topologies, such as Figure 8. In this case, one pair of machines (red and blue in this case) has multiple virtual links crossing the machine boundary. Each of these links needs to have logically separate traffic, so Mininet clustering assigns each of the links a different `key` value.

2.2 Automatic Topology Cutting

In the original Mininet, each instance of Mininet is an isolated network emulation. However, adding remote links which use GRE tunnels to cross machine boundaries allows emulated networks to be distributed across multiple machines. By running separate instances of Mininet on each machine, topologies can be hand-configured such that each machine manages a portion of a larger topology and the remote links required to connect to the rest of the cluster. This process of hand-configuration, however, is overly tedious, error-prone, and not very user-friendly, especially because it limits the interactivity of the system.

For this reason, our design for Mininet Clustering includes an algorithm for automatically cutting any topology and distributing it amongst the members of the cluster. We chose METIS [5], a set of programs for partitioning graphs which implement a faster version of the Kernighan-Lin partitioning algorithm [6]. METIS can perform k-way graph partitioning, which we can conveniently use to cut a topology and

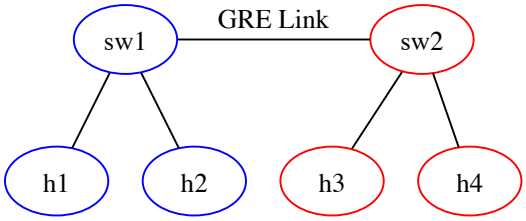

Figure 4: A simple two-switch colored topology with four virtual hosts. From the perspective of the first machine's Mininet instance, there are only two hosts and a single switch, with a GRE link.

distribute it amongst k machines. The normal version of this algorithm attempts to minimize the number of edges which cross this partition.

The algorithm METIS uses to solve this problem is a multilevel graph bisection algorithm with three phases:

1. **Coarsening Phase:** The graph G_0 is transformed into a sequence of smaller graphs $G_1, G_2, ..., G_m$ such that $|V_0| > |V_1| > |V_2| > ... > |V_m|$.

2. **Partitioning Phase:** A 2-way partition P_m of the graph $G_m = (V_m , E_m)$ is computed that partitions V_m into two parts, each containing half the vertices of G_0.

3. **Uncoarsening Phase:** The partition P_m of G_m is projected back to G_0 by going through intermediate partitions $P_m - 1$, $P_m - 2$, ... , P_1, P_0.

A topology in Mininet Clustering supports a `color` attribute for each node. This attribute indicates which member of the cluster each node will run on. Mininet Clustering defines a `make_distributed` function, which uses methods in METIS and a configuration file describing the cluster to apply apply a coloring to the graph.

Using our previous example of the simple two-switch topology in Figure 3, the `color` attribute of all nodes handled by Machine A are red, and those of Machine B are blue. The launched network would only appear to be a single switch and two hosts to the first machine. However, the switch would forward and receive traffic over a GRE link (what happens on the other side of that link is not Machine A's concern.)

3. CLUSTERING IMPLEMENTATION

Mininet Clustering was implemented with 300 lines of modifications to the Mininet 1.0.0 code. We used a Python wrapper for the METIS library [9] and wrote our own code for setting up GRE links.

The majority of Mininet Clustering's code involves converting a colored topology to a running (and distributed) Mininet. Broadly speaking, this works by having each Mininet instance compute a coloring separately. Because the algorithm is deterministic, and each instance shares the same view of the cluster and the same input topology, the colored output topology is identical.

Each cluster machine is configured with its color and a mapping of colors to IP addresses. The machine uses its color to determine what portion of the topology it is responsible for, and it uses the mapping to determine what machines it needs to create GRE links to.

```
results = []
for from_host in allHosts:
  for to_host in allHosts:
    if to_host == from_host:
      continue
    time = fromHost.median_ping(toHost,
        count = 5)
    results.append(time)
```

Figure 5: The algorithm used to measure latency.

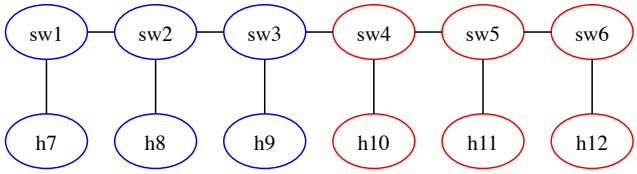

Figure 6: The representative linear topology split across two machines: blue and red. Nodes beginning with *s* are switches and those beginning with *h* are hosts.

4. EVALUATION

Our evaluations measure the difference in latency between running a Mininet topology on one computer versus splitting it across two. Figure 5 is what we used to measure the latency. While each ping is taking place, there is no other traffic on the network. This experiment, while not demonstrating the full effects of distributing a topology which once only resided locally, serves as a demonstration that the system is capable of automatically partitioning and executing network topologies, even when multiple links between machines exist.

We chose to run the experiment on what Mininet was originally designed for: a laptop. Instead of running Mininet natively, however, we decided to run it on VirtualBox with an Ubuntu 11.10 image (kernel version 3.0.0-12) supplied by the Mininet developers. This image comes with the Mininet dependencies pre-installed and configured. Further, this allowed us to emulate splitting Mininet across two machines by using one of Virtualbox's low latency local interfaces. Therefore, in this context, a *machine* is actually a virtual machine running on a MacBook Pro with a 2GHz Intel Core i7 and 4GB of 1333MHz DDR3 memory.

We evaluated latency on two different topologies: linear and tree. Figure 6 shows how the linear topology is set up. There is one split in the line of switches that crosses the machine boundary via a GRE link, and exactly 50% of the switches and virtual hosts are on each side. Each switch connects to one host. In the actual experiment, there are 20 switches and 20 hosts, 10 of each residing on each machine.

The results of the experiment on the linear topology are shown in Figure 7. The maximum latency in the graph

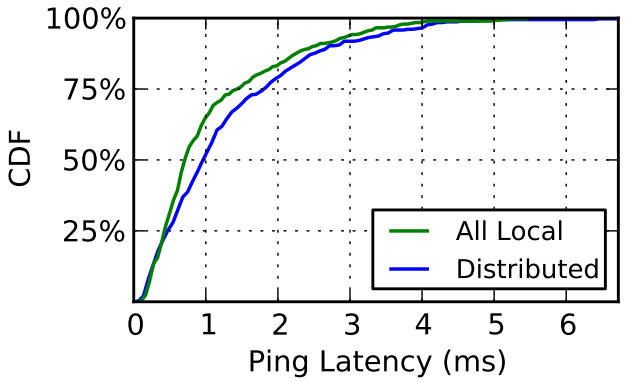

Figure 7: The experimental results on the linear topology. The green line is for when the topology is all on one machine, and the blue line is when it's split across two.

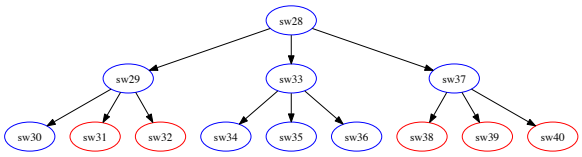

Figure 8: The tree topology used in the experiment. Each node represents a switch, split across a "blue machine" and "red machine".

is 6.0ms. The results reveal that distributing the topology across machines does not significantly affect latency. The gap in performance occurs as some normally "short" paths traverse the remote link. The distributed version exhibits a similar tail behavior as the local version, because the longest requests, which traverse 20 switches, are not hugely affected by an additional remote hop.

Figure 8 shows how the partitioning algorithm divided the nodes for the tree topology. In this diagram, each node represents a switch. Every leaf switch is connected to 3 virtual hosts. As before, the color represents which machine the switch/host is on, and the virtual hosts are on the same machine as the switch to which they connect. Unlike the last topology diagram, however, Figure 8 shows exactly what the topology was like when the experiment ran. There were 13 switches and 9 leaf switches * 3 virtual hosts each = 27 virtual hosts. In total, 40 nodes.

It is also worthwhile to note that in the split shown in the figure, red hosts cannot talk to a host on other switch without first going through a blue switch. This occurs because the partitioning algorithm attempts to balance the number of nodes between the two machines, and because of the odd numbered fan-out, it is not easy to partition the graph cleanly.

The results of the experiment on the tree topology are shown in 9. The maximum latency in the graph is 2.4ms. In this experiment, unlike in the linear topology, there is a clear flat region in the CDF of ping latencies. This occurs because for any request from a red virtual host to another

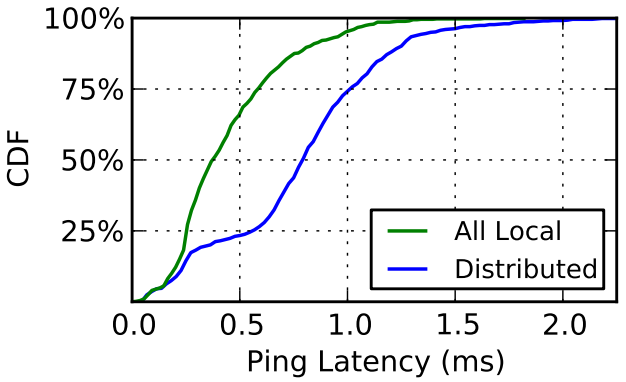

Figure 9: The components and their connections in a simple two-host Mininet network.

virtual host outside of its router, it needs to cross the machine boundary twice. This creates a region of latencies where very few requests are serviced. This suggests, that a more complex partitioning scheme may be required if this latency effect is too costly for experiments. Our partitioning scheme optimizes for the minimum edges between the graphs. A better algorithm may optimize for average number of crossings across the population of paths in the network. It could also allow for "slack" in the number of nodes running on each machine, allowing a machine to run more nodes if it would significantly improve average latency.

5. RELATED WORK

Trellis [1] is similar to MiniNet [7] in that it use container-based virtualization. It does not, however, use processes as the abstraction for virtual hosts. It is similar to our work because it uses GRE tunneling to connect multiple physical machines, except with a custom implementation the authors call Ethernet GRE. It is likely they implemented ethernet over GRE themselves because the equivalent gretap mechanism we use in the Linux kernel was not available to them at the time. Mininet with our code offers similar functionality as Trellis except that Trellis is a bit more heavyweight, not geared for OpenFlow, and doesn't provide as easy a path to deployment.

Emulab [4] offers a virtualized topology on which experiments can be run. The authors implement their own version of virtual Ethernet interfaces (veth). The paper does not discuss GRE tunneling and uses FreeBSD jails instead of Linux containers to give each host its own network namespace. Emulab's design goals are substantially different than Mininet's, but use similar techniques to accomplish them.

IMUNES [12] modifies FreeBSD to allow for multiple, simultaneous network stacks to exist within the kernel. Processes are isolated from others on their own network stacks. IMUNES's functionality is also similar to Mininet, but it does not offer the ability to hook up virtual hosts to those on other physical machines like we do with GRE tunneling.

Crossbow Virtual Wire [11] is not an emulator, but is somewhat similar to Mininet because it is another framework that uses lightweight virtualization. Crossbow uses OpenSolaris Zones which are similar to Linux containers and FreeBSD jails to isolate its network entities.

6. CONCLUSION AND FUTURE WORK

In this paper we have demonstrated that it is possible to extend Mininet's emulation capabilities beyond the single machine setup it was originally designed for. Future work includes exploring various alternative automatic partitioning strategies. Current partitioning is performed by optimizing the cut between partitions, however other metrics could be more useful depending on the network setting and the topology being emulated. Other future work will integrate our code with Mininet Hi-Fi [2]. This project enables researchers to control the bandwidth of virtual links, and we would need to extend this control to the GRE links.

Our work presented in this paper allows many more machines to be emulated in computer clusters than was possible before, increasing the feasibility of Mininet for larger or more computationally expensive experiments.

7. REFERENCES

[1] S. Bhatia, M. Motiwala, W. Muhlbauer, Y. Mundada, V. Valancius, A. Bavier, N. Feamster, L. Peterson, and J. Rexford. Trellis: A platform for building flexible, fast virtual networks on commodity hardware. In *Proceedings of the 2008 ACM CoNEXT Conference*, page 72. ACM, 2008.

[2] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container based emulation. *Proc. CoNEXT (to appear)*, 2012.

[3] S. Hanks, T. Li, D. Farinacci, and P. Traina. Generic routing encapsulation (GRE). RFC 1701, Internet Engineering Task Force, 1994.

[4] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau. Large-scale virtualization in the emulab network testbed. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 113–128. USENIX Association, 2008.

[5] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.

[6] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49:291–307, 1970 1970.

[7] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[8] 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.

[9] METIS for Python. http://pypi.python.org/pypi/metis, 2012.

[10] Open vSwitch: An open virtual switch. http://openvswitch.org/, 2012.

[11] S. Tripathi, N. Droux, K. Belgaied, and S. Khare. Crossbow virtual wire: network in a box. In *Proceedings of LISA 2009: 23rd Large Installation System Administration Conference*, page 47, 2009.

[12] M. Zec and M. Mikuc. Operating system support for integrated network emulation in imunes. In *Proc. of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS), Boston, MA*, 2004.