# The Deforestation of L2

James McCauley
UC Berkeley / ICSI

Mingjie Zhao
UESTC / ICSI

Ethan J. Jackson
UC Berkeley

Barath Raghavan
ICSI

Sylvia Ratnasamy
UC Berkeley / ICSI

Scott Shenker
UC Berkeley / ICSI

## ABSTRACT

*A major staple of layer 2 has long been the combination of flood-and-learn Ethernet switches with some variant of the Spanning Tree Protocol. However, STP has significant shortcomings – chiefly, that it throws away network capacity by removing links, and that it can be relatively slow to reconverge after topology changes. In recent years, attempts to rectify these shortcomings have been made by either making L2 look more like L3 (notably TRILL and SPB, which both incorporate L3-like routing) or by replacing L2 switches with "L3 switching" hardware and extending IP all the way to the host. In this paper, we examine an alternate point in the L2 design space, which is simple (in that it is a single data plane mechanism with no separate control plane), converges quickly, delivers packets during convergence, utilizes all available links, and can be extended to support both equal-cost multipath and efficient multicast.*

## CCS Concepts

•**Networks** → **Network protocol design;** *Link-layer protocols;*

## Keywords

L2 routing, spanning tree

## 1 Introduction

Layer 2 was originally developed to provide local connectivity while requiring little configuration. This plug-and-play property ensures that when new hosts arrive (or move), there is no need to (re)configure the host or manually (re)configure switches with new routing state. This is in contrast to IP (L3) where one must assign an IP address to a newly arriving host, and either its address or the routing tables must be updated when it moves to a new subnet. Even though L3 has developed various plug-and-play features of its own (*e.g.,* DHCP), L2 has traditionally played an important role in situations where the initial configuration, or ongoing reconfiguration due to mobility, would be burdensome. As a result, L2 remains widely used in enterprise networks and a variety of special cases such as temporary networks for events, wireless or virtual server networks with a high degree of host mobility, and small networks without dedicated support staff.

Because it must seamlessly cope with newly arrived hosts, a traditional L2 switch uses flooding to reach hosts for which it does not already have forwarding state. When a new host sends traffic, the switches "learn" how to reach the sender by recording the port on which its packets arrived. To make this flood-and-learn approach work, the network maintains a spanning tree, which removes links from the network in order to make looping impossible. The lack of loops plays two essential roles: (i) it enables flooding (otherwise looping packets would bring down the network) and (ii) it makes learning simple (because there is only one path to each host).

This approach, first developed by Mark Kempf and Radia Perlman at DEC in the early 80s [20,29], is the bedrock upon which much of modern networking has been built. Remarkably, it has persisted through major changes in networking technologies (*e.g.,* dramatic increases in speeds, the death of multiple access media) and remains a classic case of elegant design. However, users now demand better performance and availability from their networks, and this approach is widely seen as having two important drawbacks:

- The use of a spanning tree reduces the bisection bandwidth of the network to that of a single link, no matter what the physical topology is.
- When a link on the spanning tree fails, the entire tree must be reconstructed. While modern spanning tree protocol variants are vastly improved over the earlier incarnations, we continue to hear anecdotal reports that spanning tree convergence time is a recurring problem in practice, particularly in high-performance settings.[1]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

*SIGCOMM '16, August 22-26, 2016, Florianopolis , Brazil*
© 2016 ACM. ISBN 978-1-4503-4193-6/16/08. . . $15.00
DOI: http://dx.doi.org/10.1145/2934872.2934877

[1] In fact, the network administrators at our own institution have restricted the network to a tree topology so that they can turn off STP and avoid its large delays.

In this paper we present a new approach to L2, called the All conneXion Engine or AXE, that retains the original goal of plug-and-play, but can use all network links (and can even support ECMP for multipath) while providing extremely fast recovery from failures (only packets already on the wire or in the queue destined for the failed link are lost when a link goes down).[2] AXE is not a panacea, in that it does not natively support fine-grained traffic engineering, though (as we discuss later) such designs can be implemented on top. However, we see AXE as being a fairly general replacement in current Ethernets and other high-bandwidth networks where traffic engineering for local delivery is less important.

We recognize that there is a vast body of related work in this area, which we elaborate in Section 6, but now we merely note that *none* of the other designs combine AXE's features of plug-and-play, near-instantaneous recovery from failures, and ability to work on general topologies. We also recognize that redesigning L2 is not the most pressing problem in networking. However, L2 is perhaps the most widely used form of routing (in terms of the number of installations, not the number of hosts) and its performance is now seen as a growing problem (as evinced by the number of modifications and extensions vendors now deploy). What we present here is the first substantial rethinking of L2 that not only improves its performance (in terms of available bandwidth and failure recovery), but also entirely removes the need for any control plane at this layer; this is in stark contrast to STP and to many of the redesigns discussed in Section 6.

In the next section, we describe AXE's design, starting with a simplified clean design with provable correctness properties under ideal conditions, and moving on to a practical version that is more robust under non-ideal conditions. We then describe an implementation of AXE in P4 (Section 3) and extensions to support multicast (Section 4) before evaluating AXE's performance through simulation in Section 5. We end with a discussion of related work in Section 6.

## 2 Design

Traditional L2 involves two separate processes: (i) creating a tree (via STP or its variants) and (ii) forwarding packets along this tree via a flood-and-learn approach. In AXE, we only use a single mechanism – flood-and-learn – in which flooded packets are prevented from looping not with a spanning tree, but with the use of switch-based packet deduplication. Packet deduplication enables one to safely flood packets on any topology because duplicate packets are detected and dropped by the AXE switches instead of following an endless loop. This allows AXE to utilize all network links, and removes the need for a complicated failure recovery process (like STP) when links go down.

But these advantages come at the cost of a more subtle learning process. Without a spanning tree, packets can arrive along more than one path, so AXE must actively choose which of these options to learn. Furthermore, in the presence of failures, some paths may become obsolete – necessitating they be "unlearned" to make way for better ones.

To give a clearer sense of the ideas underlying AXE, we first present a *clean* version of the algorithm in Section 2.1 that has provable properties under ideal conditions. We then present a more practical version in Sections 2.2-2.4 that better addresses the non-ideal conditions found in real deployments. Both designs use the standard Ethernet *src* and *dst* addresses and additionally employ an AXE packet header with four more fields: the "learnable" flag *L*, the "flooded" flag *F*, a hopcount *HC*, and a *nonce* (used by the deduplication algorithm). We make no strong claims as to the appropriate size of the *HC* and *nonce* fields, but for the sake of rough estimation, note that if the entire additional header were 32 bits, one could allocate two bits for the flags, six for *HC* (allowing up to 64 hops), and the remaining 24 for the *nonce*. In order to maintain compatibility with unmodified hosts, we expect this header to be applied to packets at the first hop switch (which might be a software virtual switch if AXE were to be deployed in, *e.g.,* a virtualized datacenter). In addition, switches enforce a maximal *HC* (defined by the operator) to prevent unlimited looping in worst-case scenarios.

As we discuss more fully in Section 2.3, each switch uses a deduplication filter to detect (and then drop) duplicate packets based on the triplet <*src, nonce, L*>. While there are a number of ways to construct such a filter, here we use a hash-table-like data structure that can experience false negatives but no false positives (*i.e.,* no non-duplicates are dropped by the filter, but occasional duplicates may be forwarded).

The forwarding entries in a switch's learning table are indexed by an Ethernet address and contain the arrival port and the *HC* of the packet from which this state was learned (which are the port one would use to reach the host with the given address and the number of hops to reach it if a packet followed the reverse path).

Finally, note that AXE pushes the envelope for fast failure *response*, but does not make any innovation in the realm of fast failure *detection*. Instead, AXE can leverage any existing detection techniques, from higher level protocols like CFM [11] and BFD [19] to hardware-based techniques with failure detection times on the order of microseconds [7].

### 2.1 Clean Algorithm

Recall that the traditional Ethernet approach involves flooding and learning: (i) a packet is flooded when it arrives at a switch with no forwarding state for its destination address, and (ii) an arriving packet *from* a host establishes a forwarding entry *toward* that host. The approach can be so simple due to the presence of a nontrivial control plane algorithm – STP – which prunes the effective topology to a tree. Because it operates on a general topology without any control plane protocol, the clean version of AXE is slightly more complicated and can be summarized as follows:

**Header insertion and host discovery:** *When a packet arrives without an AXE header, a header is attached with* HC=1*, the* L *flag set, and the* F *flag unset. If there is no forwarding state for the source, an entry is created and the* F *flag is set.* The first step merely initializes the header; the second step is how switches learn about their attached hosts

[2] AXE was first introduced in workshop form [25]. Here we present an extended treatment.

(and the subsequent flood informs the rest of the network how to reach this host).

**Flooding:** *When a packet with the* F *flag set arrives, it is flooded out all other ports. When a packet with the* F *flag unset arrives at a switch with no forwarding state for its destination or for which the forwarding state is invalid (e.g., its link has failed), the* F *flag is set, and the packet is flooded. The flooded packets have the* L *flag set only if the flood originated at the first hop (i.e.,* HC=1*).* The flooding behavior is similar to traditional learning algorithms, with the addition of the explicit flooding and learning flags.

**Learning and unlearning:** *Switches learn how to reach the source from flooded packets with the* L *flag set, and unlearn (erase) state for the destination whenever they receive a flood packet with the* L *flag unset.* While traditional learning approaches learn how to reach the source host from all incoming packets, in AXE we can only reliably learn from packets flooded from the first hop (since packets flooded from elsewhere might have taken circuitous paths, as we discuss below). Moreover, when switches learn from flooded packets, they choose the incoming copy that has the smallest *HC*. When a flooded packet arrives with the *L* flag unset, it indicates that there is a problem reaching the destination (because the flood originated somewhere besides the first hop, as might happen with a failed link); this is why switches unlearn forwarding state when such packets arrive.

**Wandering packets:** *When the* HC *of a nonflooded packet reaches the limit, the packet is flooded (with the* F *flag set and the* L *flag unset) and local state for the destination is erased.* If the forwarding state has somehow created a loop, erasing the state locally ensures that the loop is broken. Flooding the packet (with the *L* flag unset) will cause all forwarding state to the destination host to be erased (so the next packet sent *by* that host will be flooded from the first hop, and the correct forwarding state learned).

Algorithm 1 shows pseudocode of this clean algorithm, which processes a single packet *p* at a time and consults the learning table *Table* by calling a *Lookup()* method with the desired Ethernet address. *Lookup()* returns *False* if there is no table entry corresponding to the address. The operation *Table.Learn()* inserts the appropriate updated state in the table, and *Table.Unlearn()* removes the state. *IsPortDown()* returns *True* if the output port passed to it is unavailable (*e.g.,* the link has failed). The *IsDuplicate* value (obtained from the deduplication filter) indicates whether the switch has already seen a copy of that packet (as duplicates of a packet may arrive on multiple ports if the topology contains cycles). *Output()* sends a packet via a specified port, and *Flood()* sends a packet via all ports except the ingress port (*p.InPort*).

```
if p has no AXE header then
    Add AXE header
    $p.Nonce \leftarrow NextNonce()$
    $p.HC \leftarrow 1$, $p.L \leftarrow True$
    if ! $Table.Lookup(p.EthSrc)$ then
        $p.F \leftarrow True$
        $Table.Learn(p.EthSrc, p.InPort, p.HC)$
    else
        $p.F \leftarrow False$
    end if
else
    $p.HC \leftarrow p.HC + 1$
end if

if $p.F$ then
    if ! $IsDuplicate$ then
        $Flood(p)$
    end if

    if ! $p.L$ then
        $Table.Unlearn(p.EthDst)$
    else if ! $IsDuplicate$ then
        $Table.Learn(p.EthSrc, p.InPort, p.HC)$
    else if ! $Table.Lookup(p.EthSrc)$ then
        $Table.Learn(p.EthSrc, p.InPort, p.HC)$
    else if $IsPortDown(Table.Lookup(p.EthSrc))$ then
        $Table.Learn(p.EthSrc, p.InPort, p.HC)$
    else if $Table.Lookup(p.EthSrc).HC \geq p.HC$ then
        $Table.Learn(p.EthSrc, p.InPort, p.HC)$
    end if
else if ! $Table.Lookup(p.EthDst)$ then
    $p.F \leftarrow True$, $p.L \leftarrow (HC = 1)$
    $Flood(p)$
    $Output(p, p.InPort)$
else if $IsPortDown(Table.Lookup(p.EthDst))$ then
    $p.F \leftarrow True$, $p.L \leftarrow (HC = 1)$
    $Flood(p)$
    $Output(p, p.InPort)$
else if $p.HC > MAX\_HOP\_COUNT$ then
    $p.F \leftarrow True$, $p.L \leftarrow False$
    $Flood(p)$
    $Output(p, p.InPort)$
    $Table.Unlearn(p.EthDst)$
else
    $Output(p, Table.Lookup(p.EthDst).Port)$
end if
```

Algorithm 1: The clean algorithm.

We define *ideal* conditions as when all hosts are stationary, there are no packet losses, there are no link or router failures, there are no deduplication mistakes (our mechanism ensures that there are no false positives, so this requires that there also be no false negatives), the maximal *HC* is larger than the diameter of the network, and no switch mistakenly thinks a host is directly attached when it is not. Under such conditions, we can make the following two statements about the behavior of the algorithm which hold regardless of the forwarding state currently in the switches (subject to the constraint about directly attached hosts):

**Delivery:** *Packets will be delivered to their destination.* This holds because there are only three possibilities: (i) the packet reaches the intended host following existing forwarding state (*i.e.,* it is not flooded), (ii) the packet reaches a switch without valid forwarding state and then is flooded and therefore reaches its destination, or (iii) the hopcount eventually reaches the maximal value causing the packet to be flooded which therefore reaches its destination. What cannot happen under our assumption of ideal conditions is that forwarding state on a switch delivers the packet to the wrong host (except in the case of flooding, where it reaches all hosts). Note that this line of reasoning guarantees delivery even in the non-ideal case when there are link/router failures, as long as they do not cause a partition during the time the packet and its copies are in flight.□

**Eventually shortest path routes:** *Forwarding state that is learned from undisturbed floods will route packets along shortest paths*. We call a flood from source *A* with *L* set an *undisturbed flood* if no unlearning of $A$ takes place during the time the flood has packets in transit (*e.g.,* due to other floods with destination $A$ which have the $L$ flag *un*set). New state is installed if and only if packets have both the $F$ and $L$ flags set, which happens only when packets are flooded from the first-hop switch. When intermediate switches receive multiple copies of the same packet, the ultimate state reflects the lowest hopcount needed to travel from the first-hop switch to the intermediate switch. Thus, as long as no state is erased during this process, when all copies of the flood from source $A$ with $L$ set have left the network, every switch ends up with state pointing toward an output port that has a shortest path to the destination. Any packet following this state will take a shortest path to $A$. The reason we require the flood to be undisturbed is because if some state is erased during the original flood, then the last state written may not point towards a shortest path (*i.e.,* the state that was erased may have been the state reflecting the shortest path). Note that this statement of correctness applies even if two or more flooded packets from $A$ with the $L$ flag set were in flight at the same time: since the network topology is constant under ideal conditions, all last-written state will point towards a shortest path.□

Thus, the clean design under ideal conditions, but with arbitrary initial forwarding state (subject to the constraint on attached hosts), will deliver all packets, and undisturbed floods will install shortest-path forwarding state. However, under non-ideal conditions we can make no such guarantees. Packets can be lost and routes can be far from shortest-path. Indeed, one can find examples where routing loops can be established under non-ideal conditions (though these loops will be transient, as a packet caught in such a loop will reach the maximal hopcount value, be flooded, and cause the incorrect forwarding state to be erased).

Before turning to our practical algorithm, we now explain in more detail why we need both the $L$ and the $F$ flags. Note that in traditional L2 learning, flooding and learning are completely local decisions: a switch floods a packet if and only if that switch has no forwarding state for the destination, and it learns from all packets about how to reach the source. This works because packets are either constrained to a well-formed tree (which is established via STP) or dropped (while STP is converging). In contrast, AXE switches set the $F$ flag the first time the packet arrives at a switch that has no forwarding state for the destination (or has forwarding state pointing to a dead link), and then the packet is flooded globally regardless of whether subsequent switches have forwarding state for the destination. This allows for delivery even when the forwarding state is corrupted (*e.g.,* by failures or unlearning) and there is no guarantee that following the remaining forwarding entries will deliver the packet.

While flooding is more prevalent in AXE than in traditional L2, learning is more restrictive: The clean AXE algorithm only learns from flooded packets with the $L$ flag set. This is because when packets are flooded from arbitrary locations, the resulting learned state might be misleading. Consider the network depicted in Figure 1, and imagine packets flowing from $A$ to $B$ along the path $S1$–$S2$–$S3$–$S4$–$S5$. If there is a disruption in the path, say the link $S3$–$S4$ is broken, AXE will flood packets arriving at $S3$ instead of attempting to send them down the failed link toward $S4$. Packets flooded from a failure, such as those handled by $S3$, must necessarily go *backwards* (in addition to going out all other ports), as that may be the only remaining path to the destination (as is the case in Figure 1, where after the failure of $S3$–$S4$, the only valid path to $B$ for packets at $S3$ is backward through the path $S2$–$S6$–$S4$–$S5$). One certainly does not want to learn from packets that have traveled backwards, as one could potentially be learning the *reverse* of the actual path to the destination. In this example, $S2$ would learn that $A$ is towards $S3$, which is clearly incorrect. Thus, when packets are flooded after reaching a failure, the $L$ flag is switched off, indicating that they are unlikely to be suitable for learning.

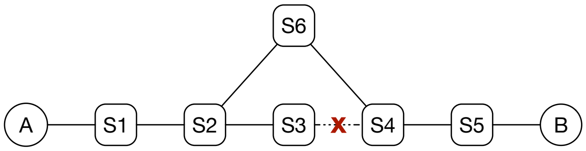


Figure 1: A network with two hosts ($A$ and $B$), six switches, and a failed link.

## 2.2 Practical Algorithm

We presented the clean algorithm to illustrate the basic ideas and show how they lead to two correctness results under ideal conditions. These ideal conditions do not hold if there is congestion, since packet losses can occur; in the clean design we liberally use packet floods when problems are encountered, which only exacerbates congestion. Thus, for our more practical approach we modify some aspects of the algorithm to reduce the number of floods, to enable learning from non-flood packets, and to give priority to flood packets. Unfortunately, our correctness results no longer hold with these modifications in place. However, simulations suggest that both the clean and the practical designs perform well under reasonable loads and failure rates, but that the practical algorithm is significantly better at dealing with and recovering from overloads or networks with high rates of failure.

The main changes from the clean design are as follows:

- When a packet exhausts its $HC$, we merely drop the packet and erase the local forwarding state (rather than flooding the packet). This reduces the number of floods under severe overloads, though the packet in question is not delivered (which violates the first correctness condition under ideal conditions).
- Switches learn from all packets with the $L$ flag set, not just flooded packets. This also reduces the number of floods, though the resulting paths are not always the shortest paths (which violates the second correctness condition under ideal conditions).
- Switches have one queue for flooded packets and another for non-flooded packets, and the flood queue is given higher priority. Because floods occur in the absence of any state or the presence of bad state, and because floods trigger learning, accelerating the delivery of floods enhances the learning of good state.

We also introduced various other wrinkles into the practical algorithm that improved its performance in simulations, such as only unlearning at the first hop and dealing with hairpinning (discussed later). While the old correctness conditions no longer hold with these changes, we *can* say that under ideal conditions (i) unless the state for an address contains a loop or is longer than the maximal HC, packets sent to it will be delivered; and (ii) the forwarding state established by undisturbed learning will enable packets to reach the intended destination, but the paths are not guaranteed to be shortest. We feel that the loosening of the correctness results is a good trade-off for the improved behavior under overload.

We later extend AXE to handle both multipath and multicast delivery, but in the next two subsections we discuss the implementation of the deduplication filter and then examine the pseudocode for the practical unipath algorithm.

## 2.3 The Deduplication Filter

Our deduplication filter provides approximate set membership with false negatives – the opposite of a Bloom filter's approximate set membership with false positives. While there are many ways to build such a filter; the approach we use is essentially a hash set with a fixed size and no collision resolution (that is, you hash an entry to find a position and then just overwrite whatever older entry may be there). Each entry contains a <*src, nonce, L*> tuple. On reception, these packet fields are hashed along with an arbitrary per-switch *salt* (*e.g.,* the Ethernet address of one of its interfaces), and the hash value is used to look up an entry in the filter's table. If the *src*, *nonce*, and *L* in the table entry match the packet, the packet is a duplicate and the filter returns *True*. If the values stored in the table entry do not match the packet, the values in the table entry are overwritten with the current packet's values, and the filter returns *False*.

Note that the response that a packet is a duplicate can only be wrong if the nonce has been repeated. We implement the nonce using a counter, but given that the nonce field has a finite size, it must eventually wrap. Thus, it is conceivable that a switch might produce a nonce such that a filter somewhere in the network still contains an entry for an older packet with the same nonce, and the possibility of this increases as the filter size increases (which is otherwise a good thing). Fortunately, we can compute the minimal amount of time required for this to occur. For example, with a 24 bit nonce space (as put forth as a possibility earlier), a 10 Gbit network transmitting min-sized packets would require around 1.16 seconds to wrap the counter (which is a relatively long time considering that the two-queue design ensures that flooded packets are delivered quickly). By timestamping entries in the filter, any entry older than this can be invalidated.

The negative response, however, can happen simply when two packets hash to the same value: the second would overwrite the first, and if another copy of the first arrived later, it would not be detected as a duplicate. We lower the probability of these false negatives by only applying packet deduplication to flooded packets (since flooded packets always "try" to loop, while non-floods only loop in the rarer case that bad state has been established), and the per-switch salt value decreases the chance that the same false negative will happen at two different switches.

## 2.4 Details of Practical Algorithm

We can now present the pseudocode for a more practical version of the AXE unipath design (Algorithm 2), in which we explicitly include invocations of the deduplication mechanism, but (for brevity) assume that the AXE header has already been added if not present and that *HC* has been incremented. The code has two phases: the first largely involves deduplication and learning/unlearning, while the second is responsible for forwarding the packet.

Some changes relative to the clean algorithm are fairly straightforward; notably the decision to drop rather than flood packets where the *HC* exceeds the maximum is embodied in lines 2-9, and the decision to learn from all packets with the *L* flag set (rather than just flooded packets) is captured in line 30. Other changes are relatively minor, such as learning even from packets with the *L* flag not set if they have a smaller *HC* (line 29), and unlearning only at first hops (lines 21-26).

A more complicated change is that concerning "hairpin turns" (line 62) where a switch has a forwarding entry pointing back the way the packet came. For an example of this, return to Figure 1 and imagine that a packet from *A* arriving at *S*3 encounters state pointing back towards *S*2. Before forwarding the packet back to *S*2, the *L* flag is unset (line 64) as *S*2 learning that the path to *A* is via *S*3 is clearly ludicrous. When the packet arrives back at *S*2, *S*2 may still have state pointing towards *S*3. While *S*2 and *S*3 could simply hairpin the packet back and forth until the hopcount reaches the maximum and the loop is resolved the same as any other loop, the combination of the forwarding state and the unset *L* flag are used to infer the presence of this special case length-two cycle and remove the looping state immediately (line 67). Note that we again make the practical decision to drop the packet and not convert it to a flood, as the existence of such a cycle is generally indicative of already adverse conditions. The other possibility is that the packet arrives back at *S*2 and *S*2 now has state which does *not* point to *S*3. Such hairpinning can arise, for example, due to multiple deduplication failures. More commonly, it is caused by our use of two forwarding queues. With two queues, a flooded packet can "pass" an already queued non-flood packet on a switch; when the non-flood one reaches the next switch, the flooded one has already changed the switch's state. For example, imagine a non-flood packet to *B* queued on *S*2 with *S*2 not yet aware that the $S3-S4$ link has failed. A learnable flood packet from *B* arrives at *S*2 (via *S*6), is placed in the high priority flood queue, and is immediately forwarded to *S*3: *S*3 learns that the path to *B* is back towards *S*2. By the time the non-flood packet finally leaves the queue on *S*2, the state on *S*2 *already* reflects the correct new path to *B* via *S*6 – as does the state on *S*3 (thus requiring the packet to take a hairpin turn).

## 2.5 Enhancements to the Design

AXE can trivially accommodate various features that L2 operators have come to rely on (*e.g.,* VLAN tagging); here we discuss three more significant ways AXE can be extended.

```
▷ * Start of first phase. *
if $p.HC > MAX\_HOP\_COUNT$ then
    ▷ Either the forwarding state loops or this is an old flood which
    ▷ the deduplication filter has never caught.
    if $!p.F$ then
        Table.Unlearn(p.EthDst)            ▷ Break looping forwarding state.
    end if
    return                                 ▷ Drop the packet.
end if

▷ Check and update the deduplication filter.
if $p.F$ then
    IsDuplicate $\leftarrow$ Filter.Contains( <p.EthSrc, p.Nonce, p.L> )
    Filter.Insert( <p.EthSrc, p.Nonce, p.L> )
else
    ▷ Non-floods aren't deduped; assume it's not a duplicate.
    IsDuplicate $\leftarrow$ False
end if

SrcEntry $\leftarrow$ Table.Lookup(p.EthSrc)
if $!IsDuplicate$ and $!p.L$ and $SrcEntry$ and $SrcEntry.HC = 1$ then
    ▷ We're seeing (for the first time) a packet which probably
    ▷ originated from this switch and then hit a failure. Since our
    ▷ forwarding state apparently points to a failure, unlearn it.
    $Table.Unlearn(p.EthDst)$
end if

if $!SrcEntry$                             ▷ No table entry, may as well learn.
  or $p.HC < SrcEntry.HC$                  ▷ Always learn a better hopcount.
  or ($p.L$ and $!IsDuplicate$)            ▷ Common case, learnable non-duplicate.
 then
    Table.Learn(p.EthSrc, p.InPort, p.HC)  ▷ Update learning table.
end if

▷ * Start of second phase. *
if $IsDuplicate$ then
    return   ▷ We've already dealt with this packet; drop the duplicate.
end if

if p.F then
    ▷ Flooded packets just keep flooding.
    Flood(p)                               ▷ Send out all ports except InPort.
    return                                 ▷ And we're done.
end if

DstEntry $\leftarrow$ Table.Lookup(p.EthDst)   ▷ Look up the output port.
if !DstEntry or IsPortDown(DstEntry.Port) then  ▷ No valid entry.
    if $!p.L$ then
        return   ▷ Packet hairpinned but is now lost. Drop and give up.
    end if

    $p.F \leftarrow True$                  ▷ About to flood the packet.
    if $p.HC = 1$ then
        ▷ This is the packet's first hop. $L$ is already set.
        Flood(p)          ▷ Flood learnably out all ports except InPort.
    else
        $p.L \leftarrow False$   ▷ Not the first hop; don't learn from the flood.
        Filter.Insert( <p.EthSrc, p.Nonce, p.L> )   ▷ Update filter.
        Flood(p)                           ▷ Sends out all ports except InPort.
        Output(p, p.InPort)                ▷ Send backwards too.
    end if
else if $DstEntry.Port = p.InPort$ then    ▷ Packet wants to hairpin.
    if $p.L$ then                          ▷ If learnable, try once to send it back.
        $p.L \leftarrow False$             ▷ No longer learnable.
        Output(p, p.InPort)
    else                                   ▷ Packet trying to hairpin twice
        Table.Unlearn(p.EthDst)            ▷ Break looping forwarding state
    end if
else
    Output(p, DstEntry.Port)               ▷ Output in the common case.
end if
```

Algorithm 2: AXE pseudocode for processing a packet $p$.

### 2.5.1 Periodic optimization

In order to make sure that non-optimal paths do not persist, switches will periodically flood packets from directly attached hosts, allowing all switches to learn new entries for it (a switch knows that it is a host's first hop because of the hopcount in its forwarding entry).

### 2.5.2 Traffic engineering

The approach AXE takes to ensure L2 connectivity is designed to be orthogonal to potential traffic engineering approaches. While some approaches aim to carefully schedule each flow in order to avoid congestion and meet other policy goals, work such as Hedera [2] showed that identifying and scheduling only elephant flows can provide substantial benefit. AXE and Hedera are complementary, and using them together only requires one extra bit in the header – a flag to indicate whether the packet should follow AXE paths or Hedera paths. In a network using both approaches, we use Hedera to compute paths for elephant flows, while mice use AXE paths. When a packet on a Hedera-scheduled flow encounters a failure, we set the extra "AXE path" flag and then route the packet with AXE. The flag is required to ensure that the packet continues to be forwarded to its destination using only AXE, as a combination of Hedera and AXE paths could produce a loop. In this way, traffic can be scheduled for efficiency, but scheduled traffic always has a guaranteed fallback as AXE ensures connectivity at all times.

### 2.5.3 ECMP

While the discussion thus far has been about unipath delivery, extending AXE to support ECMP requires only three changes: modifying the table structure, enabling the learning of multiple ports, and encouraging the learning of multiple ports. We extend the table by switching to a bitmap of learned ports (rather than a single number), and by keeping track of the nonce of the packet from which the entry was learned. Upon receiving a packet with the $L$ and $F$ flags set, if the hopcount and nonce are the same as in the table, we add the ingress port to the learned ports bitmap. If these two fields do not match, we replace (or don't replace) the entry based on much the same criteria as for the unipath algorithm. If $L$ is set and $F$ is not, we check that the hopcount and port are consistent with the current entry. If not, we replace the entry and flood the packet (to encourage a first-hop flood).

A problem with this multipath approach is that while it is easy to learn multiple paths in one direction – the originator must flood to find the recipient, and this flood allows learning multiple paths – it is not as easy to learn multiple paths in the reverse direction, as packets back to the originator will follow *one* of the equal cost paths and therefore only establish state along that single path. To address this, we need to flood in the reverse direction as well, encouraging multipath learning in both directions. This is, in fact, similar to the behavior of the "clean" algorithm discussed in Section 2.1, though our implementation here is slightly more subtle in order to integrate with the rest of the practical algorithm and to provide multiple chances to learn multiple paths given that we do not expect it to operate under ideal conditions. The key is adding another port bitmap to each table entry – a "flooded" bitmap. When a packet is going to be forwarded using an entry, if the bit corresponding to the ingress port is 0 ("hasn't yet been flooded") and the packet's hopcount is 1 (this is its first hop), we set the flooded bit for the port, and perform a flood. This is a first-hop flood, so $L$ is set, and it therefore allows learning multiple paths. The obvious downside here is some additional flooding, but the upside is that equal cost paths are discovered quickly.

## 2.6 Reasoning About Scale

In this section so far, we have focused on the details of the AXE algorithm, its properties, and what functionality it can support. Here we address another rather basic question at a conceptual (but not rigorous) level: How well does it scale?

AXE is an L2 technology that adopts the flood-and-learn paradigm. Our question is not how well a flood-and-learn paradigm can scale, because that will depend in detail on the degree of host mobility, the traffic matrix, and the bandwidth of links. Rather, our question is whether AXE's design hinders its ability to scale to large networks beyond the baseline inherent to any flood-and-learn approach. Moreover, we focus on bandwidth usage, and do not consider the impact of AXE's additional memory requirements because hardware/memory constraints will change over time. For bandwidth, the main factor that differentiates AXE from traditional L2 is the use of flooding when failures occur.

We can estimate the impact of this design choice as follows. Consider a fully utilized link of bandwidth $B$ that suddenly fails. If the average round-trip-time of traffic on the link is $RTT$, then roughly $RTT * B$ worth of traffic will be flooded before congestion control will throttle the flows. If we want to keep the overhead of failure-flooding below 1% of the total traffic, that means we can tolerate no more than $f$ failures per second, where $f = \frac{0.01}{RTT}$. If the RTT is 1ms, then the network-wide failure rate would need to be less than 10 per second. Assuming that links have MTBFs greater than $10^6$ seconds, then the traffic due to failures is less than 1% of the link for networks with less than $10^7$ links. Thus, in terms of bandwidth, AXE can scale roughly as well as any learn-and-flood approach.

# 3 P4 Implementation

We have argued – and in Section 5 we show through simulation – that AXE provides a unique combination of features: fully plug-and-play behavior, no control plane, the ability to run on general topologies, and near-instantaneous response to failures. However, if vendors were required to create a new generation of ASICs to support it, then AXE would likely be no more than an intriguing academic idea.

We think AXE can avoid this unfortunate fate because of the rise of highly reconfigurable hardware with an open-source specification language, and here we are thinking primarily of RMT [5] and P4 [18], but other such efforts may arise. In this section we discuss our implementation of AXE in P4, which we have tested in Mininet [27] using the bmv2 P4 software switch [4]. This testing verified that our implementation does, indeed, implement the protocol as expected.

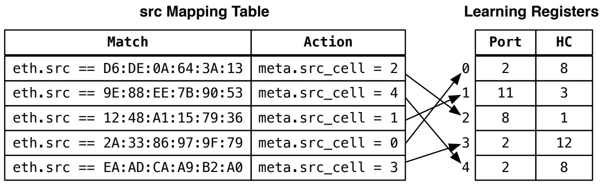
While the rest of this section delves into sometimes arcane detail, our point here is simple: once P4-supporting switches are commercially available, AXE could be deployed simply by loading a P4 implementation. While this does not assure deployment, it does radically reduce the barriers.

Unlike a traditional general-purpose programming language, P4 closely resembles the architecture of network forwarding ASICs. Programs consist of three core components: a packet parser specified by a finite state machine, a series of match-action tables similar to (but more general than) OpenFlow [26], and a *control flow function* which defines a processing pipeline (describing which tables a packet is processed by and in which order). The parser is quite general and easily implements AXE's modified Ethernet header. Thus, our primary concern was how to implement the AXE forwarding algorithm as a pipeline of matches and actions.

Putting aside the nonce, deduplication filter, and learning (discussed below), the AXE algorithm is simply a series of *if* statements checking for various special conditions. Such *if* statements can be implemented in two ways in P4: either as tables which match on various values (with the default fall-through entry acting as an *else* clause), or as actual *if* statements in the control flow function. The "bodies" of the *if* statements are implemented as P4 *compound actions*. We were able to use the slightly more straightforward latter method almost exclusively, which allowed us to structure our P4 code very similarly to the pseudocode shown above. This approach, however, is not without its caveats.

As control flow functions cannot directly execute actions, we currently have a relatively large number of "dummy" tables that merely execute a default action; the control flow function invokes these tables simply to execute the associated action (that is, the tables are always empty). If hardware performance is related to the length of the forwarding pipeline, or if there are hard limits on the number of tables (less than the 26 that we currently require), the code may need to be reorganized. Specifically, we can take the Cartesian product of nested conditionals to collapse several of them into a single table lookup. This approach can likely reduce the pipeline length dramatically, though the resulting code will surely become less readable. Whether such an optimization is necessary depends on the particular features and limitations of the associated ASIC (as well as the optimizations that the compiler backend for the target ASIC applies).

***Learning:*** P4 tables cannot be modified from the data plane – only from the control plane. This may be reasonable for a simple L2 learning switch: when a packet's source address is not in the table, the packet is sent to the switch's control plane, which creates a new table entry. Such a trip from data plane to control plane and back has a latency cost, however, and we would like to avoid it whenever possible, especially considering that AXE table entries contain not only the port, but the hopcount to reach the address' host, and keeping this hopcount information up to date is important to the algorithm. We achieve this by separating learning into two parts, as depicted in Figure 2. The first part is a table, which is populated by the control plane the first time a given Ethernet address is seen, much like a conventional L2 learning



src Mapping Table

| Match | Action |
|---|---|
| eth.src == D6:DE:0A:64:3A:13 | meta.src_cell = 2 |
| eth.src == 9E:88:EE:7B:90:53 | meta.src_cell = 4 |
| eth.src == 12:48:A1:15:79:36 | meta.src_cell = 1 |
| eth.src == 2A:33:86:97:9F:79 | meta.src_cell = 0 |
| eth.src == EA:AD:CA:A9:B2:A0 | meta.src_cell = 3 |

Learning Registers

| | Port | HC |
|---|---|---|
| 0 | 2 | 8 |
| 1 | 11 | 3 |
| 2 | 8 | 1 |
| 3 | 2 | 12 |
| 4 | 2 | 8 |

Figure 2: The first time a new MAC is seen, the packet is sent to the control plane, which adds an entry mapping the MAC to a register index. Subsequent packets are simply looked up in the mapping table, and new values for the port and hopcount can be "learned" simply by rewriting the register entirely in the data plane. A nearly identical table maps from *eth.dst* to its register index. A special port value indicates an invalid learning entry.

switch. However, instead of the table simply holding the associated port, it instead contains an index into the second part – an array of P4 registers, which are data plane state that *can* be modified by actions in the data plane. Thus, when processing a packet that requires changing learning state, it can be done at line rate entirely within the data plane, with the sole exception of the first packet (for which the control plane must allocate a register cell and add a table entry mapping the Ethernet address to it). As the P4 specification evolves, or hardware implementations support new features, it may be possible to eliminate control plane interaction entirely.

***Deduplication Filter:*** The deduplication filter is a straightforward implementation of the design discussed in Section 2.3. For reasons similar to the above, we again use an array of P4 registers to hold the filter entries rather than a P4 table (actually, we use a separate register array for each field, as the struct-like registers described in the P4 spec are not yet supported in bmv2 or its compiler). Then a P4 *field list calculation* is used to specify a hash of the appropriate header fields (the source Ethernet address, the nonce, and the *L* flag) along with a seed value stored in a P4 register and populated by the control plane at startup. The computed hash is stored in packet metadata and used to index into the filter arrays.

***Nonce:*** A switch must assign a nonce to each packet it receives from a directly-attached host. A straightforward approach is to use a single P4 register to hold a counter, though this requires that reading and incrementing the register be atomic, which may be problematic for real hardware if different ports essentially operate in parallel. Instead, we can use a nonce register per port instead of a single shared register per switch. As each port would have an independent counter, nonce allocations would no longer be entirely unique. However, this is not problematic because, as discussed in Section 2.3, the deduplication key is a tuple of <*src, nonce, L*>: for the typical case when a given host interface is only attached to a single port, the *combination* of the interface's address and a per-port nonce *will* be unique (this may preclude some types/implementations of link aggregation; however, AXE obviates some motivations for link aggregation anyway).

***Link status:*** P4 does not specify a way for the data plane to know about link liveness. Our implementation emulates this functionality by creating "port state" registers, and we manually set their values to simulate port up and down events. In a real hardware implementation, such registers could be manipulated by the control plane as it detects link state changes using whatever mechanisms the switch provides for link failure detection. We also speculate that P4-capable ASICs will have some way to query this information more directly from the data plane (without control plane involvement) for at least some failure detection mechanisms.

Our P4 implementation is not written with an eye towards efficiency on any particular P4 target, as targets are diverse and no P4 hardware target is yet available to us. Nevertheless, we see the existence of a functionally complete P4 implementation as a promising beginning.[3]

# 4 Multicast

Many L2 networks implement multicast via broadcast, with filtering done by the host NICs (sometimes with the addition of switches implementing "IGMP snooping" [6] wherein an ostensibly L2 device understands enough about L3 to prune some links). We investigated whether we could use AXE ideas to provide native support for multicast in a relatively straightforward way, and found the answer to be *yes*. We lack space to fully illuminate our design, but we sketch it here.

We try to emulate a DVMRP-like [32] multicast model, with source-specific trees for each group. While AXE's ability to safely flood makes reliable delivery easy, the design challenge is to enable rapid construction (and reconstruction) of trees in order to avoid the additional traffic overhead of unnecessary flooding. Our approach forms multicast trees by initially sending all packets for the group out all ports. Unnecessary links and switches are then pruned. When the topology changes or when new members are added to a pruned section of the tree, we simply reset the tree and reconstruct it; this avoids *maintaining* a tree as changes occur (which turns out to be quite difficult).

Multicast in AXE has four types of control messages: JOIN, LEAVE, PRUNE, and RESET. The first two are how hosts express interest/disinterest in a group to their connected switches. PRUNE is much the same as its DVMRP counterpart and is used for removing ports and switches from the tree. RESET enables a switch to indicate that something has changed which necessitates that the current tree be invalidated and rebuilt; we come back to this shortly.

Going into the algorithm in more detail, we retain much of the AXE header, but remove the *L* flag and add a "multicast generation number" field which is used for coordination. A source-group's root switch (the switch to which the source is attached) dictates a generation number for the source-group pair. Other switches that are part of the group simply track the latest number. All switches stamp all packets for this source-group (including control messages) with the latest generation number of which they are aware. If a non-root switch receives a packet for the current generation, it forwards the packet out all of the currently unpruned ports for the source-group. If the packet is for a new generation, the tree is being rebuilt; the switch moves to the new generation number and un-prunes all ports. If a packet is stamped with an old generation number, the packet is sent over all

[3] And, more broadly, we see the fact that AXE can be implemented in an ASIC-friendly language like P4 as an indicator that it may be suitable for ASIC implementation more generally – reconfigurable or otherwise.

ports; this is not optimal, but it ensures that outstanding packets from old generations are delivered even while a new generation tree is being established.

As mentioned above, when constructing a new tree, all ports are initially unpruned – this is basically the equivalent of flooding. Switches therefore potentially receive packets from the root on multiple ports, and can decide on an "upstream" port based on the best hopcount. When a switch receives a packet from the group on any port that is not its upstream port, it sends a PRUNE in response. This causes the packet's original sender to stop sending on this port. Pruning is kept consistent by ignoring PRUNEs for any generation except the current one. In this way, the initial flood-like behavior is cut down to a shortest-path tree.

When any switch notices that the tree may need to be rebuilt due to either (a) being invalid (*i.e.,* uses a link that has failed) or (b) possibly needing to expand or change shape (due to a port going up or down or a new member joining the group), the switch enters *flood mode* for the group, and may send a RESET. While a switch is in flood mode for a group, it sets the $F$ flag and floods all packets it receives for the group, disregarding whether a port has been pruned or not. The switch leaves flood mode when it sees a new, higher generation number, which indicates that the root has begun the process of building a new tree. Being in flood mode has two effects. Firstly, it makes sure that packets for the group continue to be delivered. Secondly, when the root switch sees any packet with the $F$ bit set and the current generation number, it recognizes this as meaning that some switch needs the tree to be reset. The root switch then initiates this by incrementing the generation number. While any packet can be used to reset the tree (by setting $F$), there are times when the tree should be rebuilt but no packet is immediately available (for example, when a new switch is plugged into the network). It is for these cases that the RESET message exists: they can be used to initiate a reset without needing to wait for a packet from the group to arrive (which, if the network is stable, may not be for some time).

For safety, the root switch periodically floods multicast packets even in the absence of any other indications to do so. This bounds the time that the group suffers with a bad tree if another switch is trying to reset the tree and the flood (or RESET) packets are being lost elsewhere in the network.

# 5 Evaluation

In this section, we evaluate AXE via ns-3 [28] simulations. We ask the following questions: (i) How well does AXE perform on a static network? (ii) How well does AXE perform in the presence of failures? (iii) How well does AXE cope with host migrations? (iv) How many entries are required for the deduplication filter? (v) How well does AXE recover from severe overloads? and (vi) How well does multicast work?

For some of these questions, we compare AXE to "Idealized Routing" which responds to network failures by installing random shortest paths for each destination after a specified delay. The delay is an attempt to simulate the impact of the convergence times which arise in various routing algorithms without having to implement, configure (in terms of the many constants that determine the convergence behavior), and then simulate each algorithm. Note that the time to actually compute the paths is not included in the simulated time – only the arbitrary and adjustable delay. The fact that we compute a separate and random shortest-path tree for each destination is significant: a naive shortest-path algorithm or aggregation would overlap paths significantly and not spread traffic across the network (especially in the fat tree scenario described below). This approach is not as good as ECMP, but is certainly better than a non-random approach.

We do not compare directly to spanning tree for two reasons. In terms of effectively using links, spanning tree's limitations are clear (the bisection bandwidth is that of a single link) and, as we will show, AXE is essentially as good as Idealized Routing (where the bisection bandwidth depends in detail on the network topology and link speeds). In terms of failure recovery, spanning tree is strictly worse than Idealized Routing (in that failures in spanning trees impact more flows). Thus, we view Idealized Routing as a more worthy target, providing more ambitious benchmarks against which we can compare.

## 5.1 Simulation Scenarios

We perform minute-long simulations in two different scenarios – one is a fat tree [1] with 128 hosts as might be used in a compute cluster, and the other is based on our university campus topology. For the former, we assume that links have small propagation delay (0.3us). For the latter, we assume somewhat longer propagation delays (3.5us). As we do not have specific host information for the campus topology (and it is likely to be fairly dynamic due to wireless users), we simply assign approximately 2,000 hosts to switches at random. While we would have liked to include more hosts, we limited the number in order to make simulation times manageable *for Idealized Routing* – neither our global path computation nor ns-3's IP forwarding table is optimized for large numbers of unaggregated hosts.

For each topology, we evaluate a UDP traffic load and a TCP traffic load. Although large amounts of UDP may be rare in the wild, using it as a test case helps isolate network properties (whether AXE or Idealized Routing) from the confounding aspects of TCP congestion control with its feedback loop and retransmissions. Our UDP sources merely send max-size packets at a fixed rate. For each UDP packet received, the receiver sends back a short "acknowledgment" packet to create two-way traffic (which is important in any learning scenario). For TCP traffic, we choose flow sizes from an empirical distribution [3]. In terms of UDP sending rates, in the cluster case we use a per-host rate of 100 Mbps. In the campus case, we use a per-host rate of 1 Mbps. For TCP, we pick the flow arrival rate so as to roughly match the UDP per-host sending rates. We ran simulations using both 1 Gbps and 10 Gbps links, and we omit the 10 Gbps results, which were (unsurprisingly) slightly better.

We generate traffic somewhat differently for each topology. For the cluster case, we model significant "east-west" traffic by choosing half of the hosts at random as senders, and assigning each sender an independent set of hosts as re-

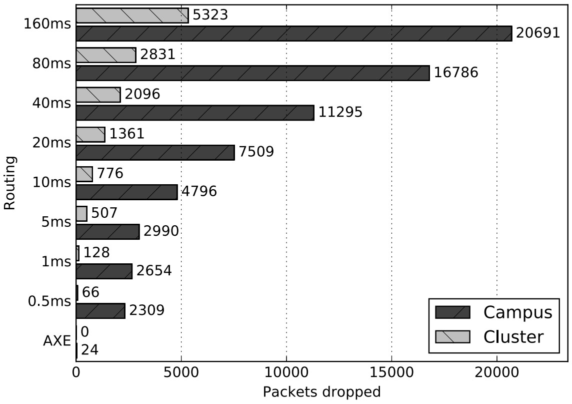


Figure 3: Comparison of unnecessary drops for AXE versus Idealized Routing with various specified convergence times.

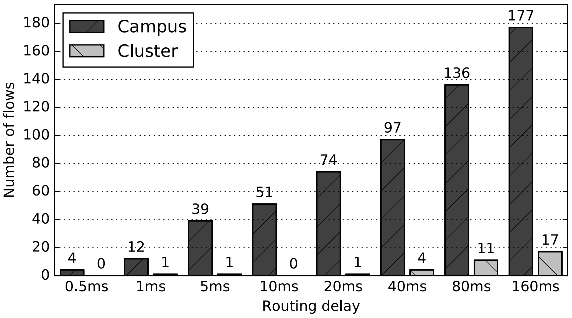


Figure 4: Number of flows where Idealized Routing suffers significantly higher FCT delay than AXE.

ceivers (each set equaling one half of the total hosts). For the campus topology, we believe traffic is concentrated at a small number of Internet gateways and on-campus servers, so all hosts share the same set of about twenty receivers.

## 5.2 Static Networks

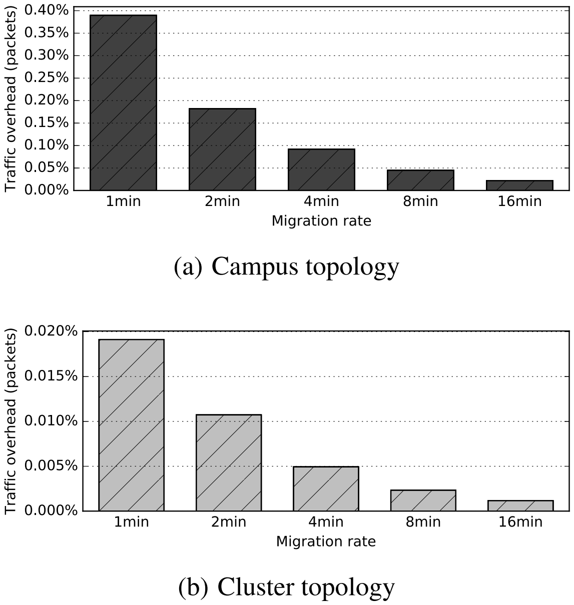
Here we show no graphs, but merely summarize the results of our simulations. In terms of setting up routes in static networks, the unipath version of AXE produced shortest path routes equivalent to Idealized Routing in both topologies, and in the cluster topology the multipath version of AXE produced multiple paths that were equivalent to an ECMP-enabled version of Idealized Routing. This is clearly superior to spanning tree, but no better than what typical L3 routing algorithms can do (and L2 protocols like SPB and TRILL that also use routing algorithms). Thus, AXE is able to produce the desired paths.

## 5.3 Performance with Link Failures

To characterize the behavior of AXE in a network undergoing failures, we "warm up" the simulation for several seconds and then proceed with one minute of failing and recovering links using a randomized failure model based on the "Individual Link Failures" in [24] but scaled to considerably higher failure rates. These failure rates represent extremely poor conditions: 24 failures over one minute for the cluster case and 193 failures over one minute for the campus case.

For simulations using UDP traffic, we looked at the number of undelivered packets, which is shown in Figure 3. In the cluster case, AXE incurs *zero* delivery failures, while Idealized Routing incurs increasingly many as the routing delay grows. In the campus case, the high failure rate and the smaller number of redundant paths leads to network partitions, and all packets sent to disconnected destinations are necessarily lost. We ignore these packets in our graph, showing only the "unnecessary" losses (packets sent to connected destinations but which routing could not deliver). We see that AXE suffers a small number of "unnecessary" losses (24), while Idealized Routing has significantly more even when the convergence delay is 0.5ms. AXE's few losses are due to overload: AXE has established valid forwarding state, but the paths simply do not have enough capacity to carry the entire load (since we were not running AXE with ECMP turned on in this experiment, Idealized Routing – which always randomizes per-destination routing – does a better job of spreading the load across all shortest paths, and so does not suffer these losses). Running this same experiment with higher capacity links, AXE achieves zero unnecessary losses.

We performed a similar experiment using TCP. However, as TCP recovers losses through retransmissions, we instead measure the impact of routing on flow completion time. We find that when comparing FCTs under AXE and Idealized Routing, either they are very close, or Idealized Routing is significantly worse due to TCP timeouts. Figure 4 shows the number of flows which appear to have suffered TCP timeouts which AXE did not (*i.e.,* have FCTs which are at least two seconds longer); there are *no* cases where the reverse is true.



(a) Campus topology

(b) Cluster topology

Figure 5: Overhead for host migrations. On average, every host migrates once per the time interval shown on the X axis. The Y axis shows the increase in total traffic.

## 5.4 Performance with Host Migrations

Migration of hosts (*e.g.,* moving a VM from one server to another or a laptop moving from one wireless access point to another) is another condition that requires re-establishing

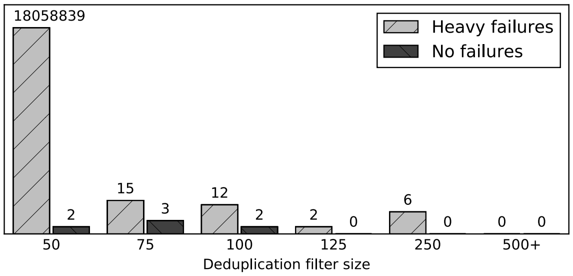


Figure 6: Effect of deduplication filter size on UDP traffic in the cluster topology with 1 Gbps links.

routes. To see how well AXE copes with migration, we run similar experiments to those in the previous section, but migrating hosts at random and with no link failures. Figure 5 shows the results of this experiment for various different rates of migration. We find that the increase in total traffic is minimal – even at the ridiculously high migration rates of each host migrating at an average rate of once per minute, the increase in traffic is under 0.44% and 0.02% for the campus and cluster topologies respectively. Note this overhead is just the overhead from AXE flooding and a gratuitous ARP; we did not model, for example, the traffic actually taken to do a virtual machine migration (though at migration rates as high as we have simulated, we would expect the AXE flooding to be vanishingly small in comparison!).

## 5.5 Deduplication Filter Size

Deduplication using our filter method is subject to false negatives – it may sometimes fail to detect a duplicate. When this happens occasionally, it presents little problem: duplicates are generally detected on neighboring switches, or at the same switch the next time it cycles around, or – in the worst case – when they reach the maximum hopcount and are dropped. However, persistent failure to detect duplicates runs the risk of creating a positive feedback loop: the failure to detect duplicates leads to more packets, which further decreases the chance of detecting duplicates.

The false negative rate of the filter is inversely correlated with the filter size, so it is important to run with filter sizes big enough to avoid melting down due to false negatives. To see how large the filter size should be, we ran simulations using filter sizes ranging between 50 and 1,600. Our simulations were a worst case, as we used the UDP traffic model (which, unlike TCP, does not back off when the network efficiency begins degrading).

Figure 6 shows the number of lost packets (which we use as evidence of harm caused by false negatives) for the cluster network with 1 Gbit links. Even under heavy failures, the number of losses goes to zero with very modest sized filters (≈500 – or even fewer for 10 Gbit links). We omit the largely similar results for the campus topology.

## 5.6 Behavior under Load

Any learning network can be driven into the ground when faced with a severe overload. Because such overloads cannot always be prevented, it is crucial that the network recovers once the problematic condition is resolved. This property follows from our design, but to verify it experimentally, we ran a simulation on the cluster topology with highly randomized traffic and a severely undersized deduplication filter. We noted the number of packets transmitted from sending hosts and the number of packets received by receiving hosts in half-second intervals, and the Y axis shows the latter divided by the former: RX/TX. Ideally one would want RX/TX to be 1, and values less than this indicate degradation of network performance. The results are shown in Figure 7.

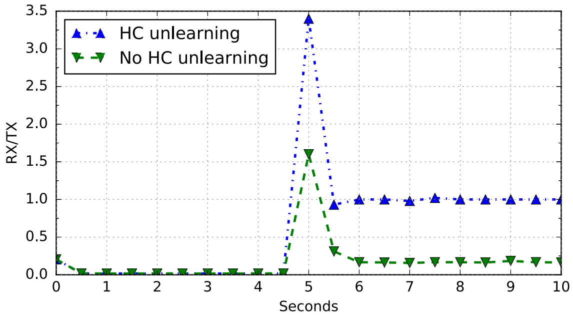


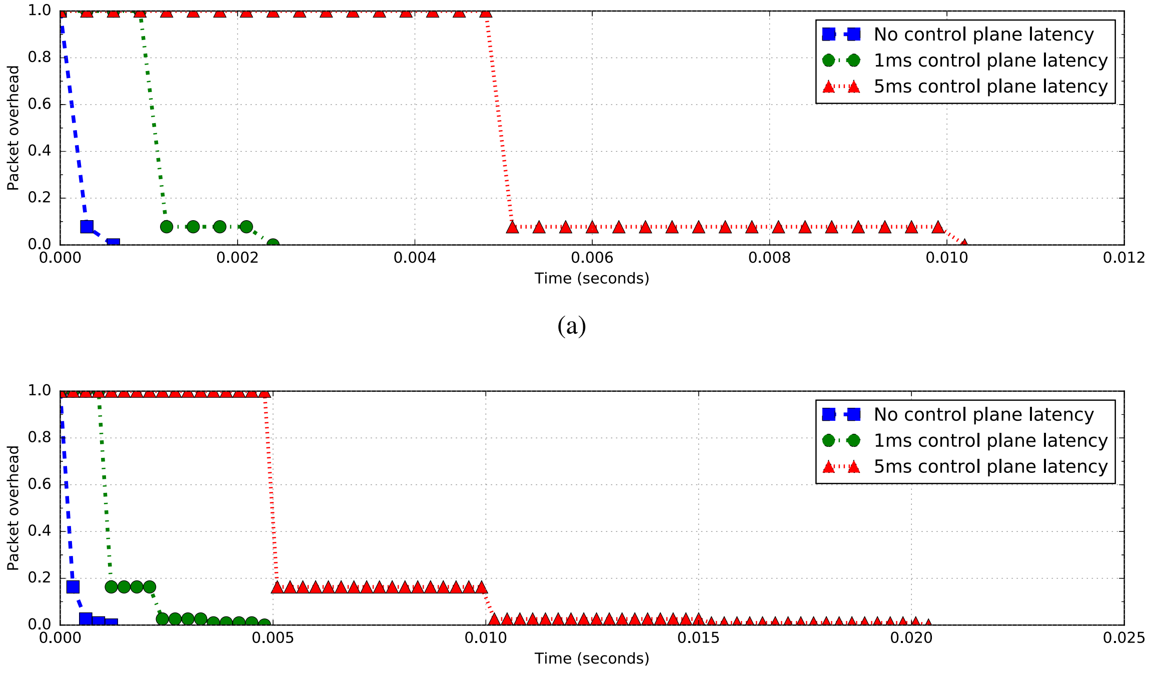
Figure 7: The ratio of received to transmitted packets in an experiment for which the first half is dramatically over-driven. We show both the true AXE algorithm which unlearns state on hopcount expiration as well as a version which does not perform unlearning.

For the first five seconds of the simulation, we generate a large amount of traffic (far more than the links and deduplication filter can accommodate). This drives AXE into a useless state where packets are flooding, dropping, being delayed, and are not reliably deduplicated or learned from. Indeed, the fraction of this traffic that is delivered successfully is negligible. At five seconds, we reduce the traffic to a manageable level. We see that following a spike in delivery (as the queues that built up in the first half of the experiment drain), AXE quickly reaches the expected equilibrium.

We also verified that one of AXE's safety mechanisms has the expected effect. Specifically, when the hopcount reaches its limit for a non-flood packet, the final switch removes (unlearns) state for the packet's destination. In this way, any future packets to that destination will not follow the looping path, but will instead find themselves with no forwarding state and be flooded. To witness this in action, we disable AXE's hopcount expiration unlearning. This results in a dramatic drop in RX/TX ratio, because bad (looping) paths established in the first part of the experiment are never repaired.

## 5.7 Multicast

While traditional approaches to multicast maintain trees through incremental prunes and grafts, our AXE multicast design rebuilds the tree from scratch every time there is a change. Rebuilding a tree requires flooding packets and then letting the tree be pruned back. Whether this approach is reasonable or not depends on whether periodically switching back to flooding is overly wasteful. While networks can clearly withstand the occasional flooded packet (broadcasts for service discovery, and so on), the danger with multicast is that rebuilding the tree during a high volume multicast transmission (such as a video stream) may result in a large number of packets being flooded. To examine this case, we simulated



(a)

(b)

Figure 8: Convergence of a multicast tree using AXE while a source transmits a simulated 40Mbps video stream on the campus (a) and cluster (b). Each mark in the graph is a packet transmitted from the source. The X axis is time, with 0 being the transmission time of the first packet following a tree reset. The Y axis shows the normalized packet overhead: 1.0 is when the packet is being flooded, and 0.0 is when the packet is being sent only along the final converged tree.

a transmission of data at 40Mbps (a rather extreme case – equivalent to a high-quality 1080p Blu-ray Disc) and examined the convergence of AXE's multicast after triggering a reset of the multicast tree. We repeated the experiment several times on both topologies and with group membership between 5% and 40%, and show a representative example of a 5% run in Figure 8. The graphs show the overhead of extra traffic, where a value of 1 indicates sending as much traffic as a flood, and an overhead of 0 indicates sending as much traffic as a shortest-path tree.

In the AXE multicast algorithm, the data plane recognizes that a PRUNE should be sent, but the control plane is ultimately responsible for creating the PRUNE message. Interactions between the control and data planes, however, are not especially fast in terms of packet timescales, so we modeled 1ms and 5ms latencies along with 0ms for comparison.[4] We find that even in the worst case, AXE converges quickly: the overhead has dropped to less than 20% by about 5ms and is either converged (on the campus topology) or negligible (on the cluster topology) by about 10ms – and even at the high rate of 40 Mbps, only 34 packets are sent during this 10ms.

It is worth noting that the convergence time is almost entirely dictated by the control plane latency: with no control plane latency, in the two experiments we show, the trees have converged by the time the third or fifth packet is sent. Indeed, for experiments we ran with a larger number of group members, it had often converged by the second packet (the final tree is larger when there are more members, and it therefore takes fewer prunes to converge to it).

## 6 Related Work

Since the introduction of Ethernet, there have been efforts to improve upon it, both in terms of its operational model and its performance. AXE represents a new direction in this body of literature, providing many advantages traditionally associated with routing without sacrificing the simple plug-and-play nature of traditional MAC learning.

There have been a wide variety of enhancements to Spanning Tree Protocol. Rapid Spanning Tree [13] improves upon STP with faster link and failure recoveries. Additionally, there have been numerous enhancements that build multiple spanning trees in a single network to improve utilization and redundancy. These include MSTP, MISTP [15], PVST, PVST+ [14], and others. AXE dispenses with spanning trees altogether, allowing it to achieve instantaneous failure recovery and achieve short paths that utilize all links.

In datacenter networks, one trend that addresses many concerns surrounding STP is to abandon L2 entirely, instead running IP all the way to the host [1, 9]. This approach performs well (similar to the Idealized Routing we evaluated against in Section 5), though not as well as AXE in terms of how quickly it recovers from failures. Work along the same lines has used highly specialized Clos topologies coupled with an

[4] OpenFlow switches have had latencies measured from 1.72ms up to 8.33ms for expensive table insertions [10]. AXE's multicast control plane running directly on the switch would avoid OpenFlow-specific overheads and requires no expensive table update, so 5ms seems an outside estimate.

SDN control plane to achieve truly impressive results in a datacenter context [31]. We note that none of these techniques achieve AXE's plug-and-play functionality, and all of them require specially designed network topologies. AXE, on the other hand, works on arbitrary topologies without issue, making it ideally suited for less uniform environments.

Also in the datacenter context, F10 [23] achieves impressive results by co-designing the topology, routing, and failure detection. While it shares one of AXE's primary goals (fast failure response), the highly-coupled approach is starkly different. VL2 [9] maintains L2 semantics for hosts, but does so using a directory service rather than flood-and-learn, and, again, is designed with particular topologies in mind.

Recently there has been interest in bringing the techniques of routing to L2 through technologies like SPB [12] and TRILL [30]. These protocols use link state routing to compute shortest paths between edge switches, thus inheriting the advantages (and limitations) of L3 routing protocols.

In some ways, AXE is similar to Dynamic Source Routing [17], 802.2 Source Route Bridging [16], and AntNet [8]. These schemes all send special "explorer packets" looking for destinations, collecting a list of switches they have passed through. Upon reaching the destination, a report is sent back to the source which can use the accumulated list of switches to build a path. AXE differs most fundamentally in that it does not use or create any special routing messages – everything needed to establish routes is contained in the AXE packet header; rather than have the destination explicitly return a special routing packet, it relies on the destination to create a packet in the return direction in the normal course of communication (*e.g.,* a TCP ACK). This difference also applies to failure recovery: while DSR has another special type of routing message to convey failure information and induce the source to "explore" for new routes again, this too takes place with plain data packets in AXE. An outcome of all this is that AXE is simultaneously routing *and* delivering user data – there is no distinction (or latency) between them. A further difference is that AXE does not keep work-in-progress routing state in special packets, and instead uses per-switch learning, requiring an alternate loop-prevention strategy. While an SRB or an AntNet switch can identify a looped packet because its own identifier already exists in the packet's list of hops, AXE packets contain no such list; thus, the switch must "remember" having seen the packet.

Like AXE, Failure Carrying Packets (FCP) [21] minimizes convergence times by piggybacking information about failures in data packets. In FCP, each switch has a (more-or-less) accurate map of the network, and each packet contains information about failed links in the map that it has encountered. This is sufficient for an FCP switch to forward the packet to the destination if any working path is available. AXE has no such network map and so its strategy for failed packets is simply to flood the packet and remove faulty state so that it can be rebuilt (via learning).

Data-Driven Connectivity (DDC) [22] also uses data packets to minimize convergence times during failure. It uses a global view of the network to construct a DAG for each destination. When failures occur, packets are sent along the DAG in the wrong direction, which signals a failure to the receiving switch which can then (via flipping the directions of adjacent edges) find a new path to the destination. Thus, FCP, DDC, and AXE all use packets to signal failures, but whereas FCP and AXE use actual bits in the header, the signaling in DDC is implicit. Unlike AXE, which generally builds shortest paths, paths in DDC may end up arbitrarily long. And similar to FCP and dissimilar to AXE, it relies on some sort of control plane to build a global view of the network.

Despite this large set of related efforts, *none* combine all of AXE's features: plug-and-play, near-instantaneous recovery from failures,[5] and ability to work on general topologies. Thus, we see AXE as occupying a useful and unique niche in the networking ecosystem.

# 7 Conclusion

Ultimately, our goal is to develop AXE as a general-purpose replacement for off-the-shelf Ethernet, providing essentially instantaneous failure recovery, unicast that makes efficient use of bandwidth (not just short paths, but also ECMP-like behavior), and direct multicast support – while retaining Ethernet's plug-and-play characteristics and topology agnosticism. We are not aware of any other design that strikes this balance. While we do not see AXE as a contender for special-purpose high-performance datacenter environments (where plug-and-play is largely irrelevant), in most other cases we see it as a promising alternative to today's designs.

# 8 Acknowledgments

We wish to thank the anonymous reviewers and especially our shepherd, Brad Karp, for their thoughtful feedback.

This material is based upon work supported by sponsors including Intel, AT&T, and the National Science Foundation under Grants No. 1420064, 1216073, and 1139158.

# 9 References

[1] Al-Fares, M., Loukissas, A., and Vahdat, A. A Scalable, Commodity Data Center Network Architecture. In *Proc. of SIGCOMM* (2008).

[2] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proc. of NSDI* (2010).

[3] Benson, T., Akella, A., and Maltz, D. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. of ACM Internet Measurement Conference (IMC)* (2012).

[4] P4 Behavioral Model. https://github.com/p4lang/behavioral-model.

[5] Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Proc. of SIGCOMM* (2013).

[5] That is, limited only by failure detection time and the time required for the switch to change forwarding behavior – *not* by network size, communication with other switches, etc.

[6] CHRISTENSEN, M., KIMBALL, K., AND SOLENSKY, F. Considerations for Internet Group Management Protocol (IGMP) and Multicast Listener Discovery (MLD) Snooping Switches. RFC 4541 (Informational), 2006.

[7] DATASHEET, TEXAS INSTRUMENTS. DP83867IR/CR Robust, High Immunity 10/100/1000 Ethernet Physical Layer Transceiver. http://www.ti.com/lit/ds/symlink/dp83867ir.pdf, 2015.

[8] DI CARO, G. A., AND DORIGO, M. Two Ant Colony Algorithms for Best-Effort Routing in Datagram Networks. In *Proc. of the IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)* (1998).

[9] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. VL2: A Scalable and Flexible Data Center Network. In *Proc. of SIGCOMM* (2009).

[10] HE, K., KHALID, J., GEMBER-JACOBSON, A., DAS, S., PRAKASH, C., AKELLA, A., LI, L. E., AND THOTTAN, M. Measuring Control Plane Latency in SDN-enabled Switches. In *Proc. of the ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)* (2015).

[11] IEEE STANDARDS ASSOCIATION. 802.1ag-2007 - IEEE Standard for Local and Metropolitan Area Networks Virtual Bridged Local Area Networks Amendment 5: Connectivity Fault Management. http://standards.ieee.org/findstds/standard/802.1ag-2007.html.

[12] IEEE STANDARDS ASSOCIATION. 802.1aq-2012 - IEEE Standard for Local and metropolitan area networks–Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks–Amendment 20: Shortest Path Bridging. https://standards.ieee.org/findstds/standard/802.1aq-2012.html.

[13] IEEE STANDARDS ASSOCIATION. 802.1D-2004 - IEEE Standard for Local and metropolitan area networks: Media Access Control (MAC) Bridges. http://standards.ieee.org/findstds/standard/802.1D-2004.html.

[14] IEEE STANDARDS ASSOCIATION. 802.1Q-2014 - IEEE Standard for Local and metropolitan area networks–Bridges and Bridged Networks. http://standards.ieee.org/findstds/standard/802.1Q-2014.html.

[15] IEEE STANDARDS ASSOCIATION. 802.1s-2002 - IEEE Standards for Local and Metropolitan Area Networks - Amendment to 802.1Q Virtual Bridged Local Area Networks: Multiple Spanning Trees. http://standards.ieee.org/findstds/standard/802.1s-2002.html.

[16] IEEE STANDARDS ASSOCIATION. 802.2-1989 - IEEE Standard for Information Technology - Telecommunications and Information Exchange Between Systems - Local and Metropolitan Area Networks - Specific Requirements - Part 2: Logical Link Control. http://standards.ieee.org/findstds/standard/802.2-1989.html.

[17] JOHNSON, D. B. Routing in Ad Hoc Networks of Mobile Hosts. In *Proc. of Workshop on Mobile Computing Systems and Applications (WMCSA)* (1994).

[18] JOSE, L., YAN, L., VARGHESE, G., AND MCKEOWN, N. Compiling Packet Programs to Reconfigurable Switches. In *Proc. of NSDI* (2015).

[19] KATZ, D., AND WARD, D. Bidirectional Forwarding Detection (BFD). RFC 5880 (Proposed Standard), 2010.

[20] KEMPF, M. Bridge Circuit for Interconnecting Networks, 1986. US Patent 4,597,078.

[21] LAKSHMINARAYANAN, K., CAESAR, M., RANGAN, M., ANDERSON, T., SHENKER, S., AND STOICA, I. Achieving Convergence-free Routing Using Failure-carrying Packets. In *Proc. of SIGCOMM* (2007).

[22] LIU, J., PANDA, A., SINGLA, A., GODFREY, B., SCHAPIRA, M., AND SHENKER, S. Ensuring Connectivity via Data Plane Mechanisms. In *Proc. of NSDI* (2013).

[23] LIU, V., HALPERIN, D., KRISHNAMURTHY, A., AND ANDERSON, T. F10: A Fault-Tolerant Engineered Network. In *Proc. of NSDI* (2013).

[24] MARKOPOULOU, A., IANNACCONE, G., BHATTACHARYYA, S., CHUAH, C.-N., AND DIOT, C. Characterization of Failures in an IP Backbone. In *Proc. of INFOCOM* (2004).

[25] MCCAULEY, J., SHENG, A., JACKSON, E. J., RAGHAVAN, B., RATNASAMY, S., AND SHENKER, S. Taking an AXE to L2 Spanning Trees. In *Proc. of HotNets* (2015).

[26] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling Innovation in Campus Networks. *CCR 38*, 2 (2008).

[27] Mininet. http://mininet.org/.

[28] ns-3. http://www.nsnam.org/.

[29] PERLMAN, R. An Algorithm for Distributed Computation of a Spanning Tree in an Extended LAN. In *Proc. of SIGCOMM* (1985).

[30] PERLMAN, R., EASTLAKE, D., DUTT, D., GAI, S., AND GHANWANI, A. Routing Bridges (RBridges): Base Protocol Specification. RFC 6325 (Proposed Standard), 2011.

[31] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P., ET AL. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proc. of SIGCOMM* (2015).

[32] WAITZMAN, D., PARTRIDGE, C., AND DEERING, S. Distance Vector Multicast Routing Protocol. RFC 1075 (Experimental), 1988.