Taking an AXE to L2 Spanning Trees

James McCauley  
UC Berkeley / ICSI

Alice Sheng  
UC Berkeley

Ethan J. Jackson  
UC Berkeley

Barath Raghavan  
ICSI

Sylvia Ratnasamy  
UC Berkeley

Scott Shenker  
UC Berkeley / ICSI

ABSTRACT

I think that I shall never see  
a structure more wasteful than a tree.  
Most links remain idle and unused  
while others are overloaded and abused.  
And with each failure comes disruption  
caused by the ensuing tree construction.  
Thus, L2 must discard its spanner,  
requiring flooding in a different manner.  
For the tree’s fragile waste to be abated,  
trim no branches and detect packets duplicated.

(With apologies to Radia Perlman and Joyce Kilmer.)

Categories and Subject Descriptors
C.2.2 [Computer-Communication Networks]: Network 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 (layer 3) where one must assign an IP address to newly arriving hosts, and when a host moves to a new subnet, either its address or the routing tables must be updated. Thus, even though L3 has developed various plug-and-play features of its own (e.g., DHCP), L2 has traditionally and continues to play an important role in situations involving host mobility where such reconfiguration would be burdensome.

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 this host by recording the port on which the host’s 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 (which in turn makes learning simple because there is only one path to each host from any given location).

This approach, first developed by Mark Kempf and Radia Perlman at DEC in the early 80s [10,17], is the bedrock upon which much of modern networking has been built, and it has persisted through major changes in networking technologies (e.g., dramatic increases in speeds, the death of multiple access media). However, users now demand better performance and availability from their networks, and this approach is widely seen as having two important drawbacks. First, the use of a spanning tree leaves many of the network links unused, and in fact the bisection bandwidth is merely the bandwidth of a single link. Second, whenever one of the links on the spanning tree fails, the entire tree must be reconstructed; while modern spanning tree protocol variants (e.g., RSTP) are vastly improved over the earlier incarnations, we continue to hear anecdotal reports that in practice spanning tree convergence times are an ongoing problem.

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) and provides 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). AXE is not a panacea, in that it does not natively support fine-grained traffic engineering to deal with elephant flows (as in [2]), though (as we discuss later) such designs can be implemented on top of AXE. However, we see AXE as being a fairly general replacement for current Ethernets and other high-bandwidth networks where traffic engineering for local delivery is not required.

We recognize that there is a vast body of related work in
We now turn to AXE’s design, starting with an overview, then delving into more detail with a lengthy piece of pseudocode.

2 Design

We now turn to AXE’s design, starting with an overview, then delving into more detail with a lengthy piece of pseudocode.

2.1 Overview

AXE follows traditional L2 in using a flood-and-learn approach, but with two major changes. First, flooding in AXE avoids loops not with a spanning tree, but with the use of switch-based packet deduplication (which we describe below), which is enabled by having each packet carry a nonce that, along with its source address, renders it unique over end-to-end timescales. Second, while AXE uses learning (where packets from a host establish forwarding entries toward that host), AXE’s learning process must also compensate for the lack of a spanning tree and “unlearn” failed paths (which helps reestablish the appropriate routing state after a failure).

Even with these two deviations from classic L2, AXE’s operation is conceptually quite simple. When a packet arrives at a switch which does not have forwarding state for it, or does have forwarding state but it points towards a failed link, the packet is flooded. Packet deduplication eliminates looping regardless of topology, so AXE does not need to wait for a complicated failure recovery process when links go down; packets are merely flooded to find new paths around failures. These floods traverse all links, so many routes are explored, including all shortest path routes (assuming no packet drops, and where by shortest we mean lowest delay given current conditions). Thus, in the ideal case, AXE routes all packets along shortest paths, which enables far better use of network resources than spanning tree. Moreover, when extended to the ECMP case (as we explain later), AXE can fully exploit multiple equal cost paths.

Of course, nothing is quite this simple. We next describe a few other aspects of the design, then present the pseudocode for its implementation of unipath delivery.

AXE Packet Header: In addition to standard Ethernet fields (only two of which we explicitly use in AXE: the src and dst addresses), the AXE header additionally contains two flags (the “learnable” flag L and the “flooding” flag F), a hop count (HC), and a nonce. In order to maintain compatibility with unmodified hosts, we expect this header to be applied to packets at the first hop switch (in virtualized datacenters, this may well be a virtual switch). When created, the L flag is set, the F flag is unset, the hop count is zero (though it is incremented before leaving the first hop), and the nonce is set to the current value of a counter (which is then incremented). We make no strong claims as to the appropriate size for each of these fields, but note that if the entire 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. As we discuss below, it is desirable for a <nonce,src> tuple to uniquely identify packets in-flight (or copies thereof); by the time we have wrapped the nonce space, we would like to be sure that any earlier packets with the same nonce have left the network.

Queueing: Packets are forwarded using one of two queues, depending on whether the packet has the flood (F) bit set. The flood packet queue gets high priority, ensuring that packets to destinations without routing state are delivered quickly (so routes can be learned from the response quickly).

Failure Detection: AXE does not implement its own failure detection mechanism, but leverages existing physical detection techniques or BFD. It is true that in some current deployments the delay in detecting failures is far greater than the time it takes for routing to repair them. However, there are known techniques for rapidly detecting hardware failures (e.g., as in SONET), so in this paper we are focused on rapid recovery (for which there are no current proposals for topology-agnostic mechanisms that support plug-and-play).

Packet Deduplication: We eliminate duplicate packets using what we call a wilt filter (because it provides approximate set membership with false negatives – the opposite of a Bloom filter’s approximate set membership with false positives). The wilt filter is essentially a hash table, where each entry contains a <src, nonce, L> tuple. On reception, a packet’s src, nonce, and L 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, which is unlikely given the size of the nonce field we are using. The negative response, however, can hap-

1 In addition, there is a wealth of work on new transport mechanisms (e.g., RSTP, MSTP, MISTP, PVST), (ii) reshape L2 to use normal routing and provide plug-and-play via mappings to translate addresses to destination switches (e.g., SPB, TRILL), (iii) avoid using L2 by using L3 almost exclusively (as in many high-performance datacenter environments), (iv) optimize designs for special topologies and/or use cases (as in F10 [13], VL2 [6], and DSR [7]), and (v) achieve rapid failure recovery (e.g., F10 [13], FCP [11], DDC [12]).

2 A link is traversed only in one direction if the packet arrives at the other end before the packet destined in the other direction has been enqueued. If not, then the packet traverses the link in both directions, but neither copy of the packet is forwarded further.

3 With a 24 bit nonce space, it would take a 10 Gbit network transmitting min-sized packets over 1.16 seconds to wrap the counter, which seems more than sufficient.
We want to learn short paths (AXE algorithm as would be implemented for handling packets on a switch for unipath routing. This pseudocode is fairly easy to understand, and some of the more nuanced aspects of the actual algorithm. One might ask: why is this, given that L2 learning algorithms are completely straightforward? The reason is that AXE must cope with two issues that do not exist in standard L2 learning: the existence of multiple paths (because there is no spanning tree), and the need to react quickly to failures (which requires unlearning some routes). We want to learn short paths (i.e., select wisely from the multiple possible outgoing ports) but also respond quickly when paths change (which requires recognizing when old paths are no longer valid). Thus, there is a tension between finding good paths (always select the shortest path you’ve seen) and finding new paths (always select the most recent path you’ve seen), and our code tries to walk the fine line between them.

The code is largely divided into two phases: an ingress portion largely involving deduplication and learning/unlearning, and an egress portion responsible for forwarding a packet towards its destination. In addition to the header fields and deduplication interface, the code utilizes (as do all learning algorithms) a learning table that associates an address with a port on which that address was seen (and, in our case, also includes the hop count of the packet from which the entry was learned).

To ease understanding of the pseudocode, it is useful to have some sense of how the “learnable” or L header flag is used (which, as a reminder, defaults to “on”). In general, when a packet arrives at a switch, we wish to learn that the source of the packet can be reached via the ingress port. However, there are cases where this is a bad idea. For example, when a packet reaches a failure in AXE, it is typically flooded (line 57) – this is how we achieve very high rates of delivery even during failures. However, when being flooded from a failure, a packet must go backwards (line 58), as it may be that the only remaining path to the destination lies back toward the source. As a packet travels backwards, one certainly does not wish to learn from this packet, as one would be learning the entirely incorrect direction. Thus, when packets are flooded after reaching a failure, the L bit is switched off (line 55), indicating that they are unlikely to be suitable for learning. For the same reason, the L bit is switched off when a packet makes a hairpin turn (line 62) – when it reaches a switch that has a forwarding entry pointing back the way the packet came (a situation that can occur due to the two queue design when a flooded packet “passes” 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).

Second, we institute an approximate global quota on the rate of floods. As flooded packets appear on every link, each switch can simply count the bytes in each non-duplicate flood packet it receives; all switches should be computing approximately the same count. When this number exceeds a threshold, a switch can halt generating new floods until the flood load is again acceptable. This ensures that the network does not enter a state where floods overwhelm all other traffic.

2.2 Algorithm

In Figure 1, we show a pseudocode implementation of the AXE algorithm as would be implemented for handling packets on a switch for unipath routing. This pseudocode is fairly lengthy, and even so omits some of the more nuanced aspects of the actual algorithm. One might ask: why is this, given that L2 learning algorithms are completely straightforward? The reason is that AXE must cope with two issues that do not exist in standard L2 learning: the existence of multiple paths (because there is no spanning tree), and the need to react quickly to failures (which requires unlearning some routes). We want to learn short paths (i.e., select wisely from the multiple possible outgoing ports) but also respond quickly when paths change (which requires recognizing when old paths are no longer valid). Thus, there is a tension between finding good paths (always select the shortest path you’ve seen) and finding new paths (always select the most recent path you’ve seen), and our code tries to walk the fine line between them.

The code is largely divided into two phases: an ingress portion largely involving deduplication and learning/unlearning, and an egress portion responsible for forwarding a packet towards its destination. In addition to the header fields and deduplication interface, the code utilizes (as do all learning algorithms) a learning table that associates an address with a port on which that address was seen (and, in our case, also includes the hop count of the packet from which the entry was learned).

To ease understanding of the pseudocode, it is useful to have some sense of how the “learnable” or L header flag is used (which, as a reminder, defaults to “on”). In general, when a packet arrives at a switch, we wish to learn that the source of the packet can be reached via the ingress port. However, there are cases where this is a bad idea. For example, when a packet reaches a failure in AXE, it is typically flooded (line 57) – this is how we achieve very high rates of delivery even during failures. However, when being flooded from a failure, a packet must go backwards (line 58), as it may be that the only remaining path to the destination lies back toward the source. As a packet travels backwards, one certainly does not wish to learn from this packet, as one would be learning the entirely incorrect direction. Thus, when packets are flooded after reaching a failure, the L flag is switched off (line 55), indicating that they are unlikely to be suitable for learning. For the same reason, the L flag is switched off when a packet makes a hairpin turn (line 62) – when it reaches a switch that has a forwarding entry pointing back the way the packet came (a situation that can occur due to the two queue design when a flooded packet “passes” 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).

The counter case is when a packet is simply following a path or is flooded from its first hop (line 53). In such cases, the L bit can (and should) be left in its default (enabled) state. Also note that the L bit is included in the wilt filter entries (lines 11, 12, and 56). This is so that a packet which is intentionally traveling backward (e.g., in response to a failure) is not seen as a duplicate and dropped.

A final note is that the pseudocode has separate operations to check the wilt filter and to update it, though these are a single operation in our abstract description in Section 2.1. We separate them here as there is a case where we update the filter but need not check for duplication (line 56).

While the pseudocode and discussion thus far has been on 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 bit set, rather than simply always replacing the existing entry, if the hop count and nonce are the same, we include the ingress port in the learned ports bitmap. If these two fields do not match, we replace (or don’t replace) the entry based on the same criteria as for the unipath algorithm.

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. To do so, we add 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 hop count 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.

3 Evaluation

In this section, we evaluate AXE by attempting to answer three questions about whether and how well AXE works, primarily by using simulations performed in ns-3 [16]: (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 perform in the presence of failures? (iii) How many entries are required for the filter?

For some of these, we compare AXE to “Idealized Routing” which responds to network failures by executing an all-pairs shortest path algorithm after a specified delay (and has a separate routing entry for each host). This 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.

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 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.

3.1 Simulation Scenarios

We perform minute-long simulations in two quite different scenarios – a datacenter case and a university campus case. The former is a fat tree [1] with 128 hosts as might be used in a small virtualized cluster. For this experiment, we assume that links have small propagation delay (0.3us). Our other scenario is a topology modeled after that of our university campus, and we assume somewhat longer propagation delays (3.5us). As we do not have specific host information for this 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

```plaintext
1:  We begin with the ingress phase.
2:  if p.HC > MAX_HOP_COUNT then
3:     Either the forwarding state loops or this is an old flood which
4:     the filter has never caught.
5:  Table.Unlearn(p.EthDst)  // Break looping forwarding state.
6:  return true  // Drop the packet.
7:  end if
8:
9:  check and update the deduplication filter.
10: if p.F then
13: else
14:     // Non-floods aren’t deduplicated assume it’s not a duplicate.
15:     IsDuplicate ← false
16:  end if
17:
18:  SrcEntry ← Table.Lookup(p.EthSrc)
19:  if !IsDuplicate and p.L and SrcEntry and SrcEntry.HC == 1 then
20:     // We’re seeing (for the first time) a packet which probably originated
21:     // from this switch and then hit a failure. Since our forwarding state
22:     // apparently led the packet to a failure; unlearn it.
23:     Table.Unlearn(p.EthDst)
24: end if
25:
26:  if !SrcEntry  // No table entry, may as well learn.
27:    or p.HC < SrcEntry.HC  // Always learn a better hop count.
29: then
31: end if
32:
33:  // Now, the egress phase.
34: if IsDuplicate then
35:     return true  // We’ve already dealt with this packet; drop the duplicate.
36: end if
37:
38:  if p.F then
39:     // Flooded packets just keep flooding.
40:     Flood(p)  // Send out all ports except InPort.
41:     return true  // And we’re done.
42: end if
43:
44:  DstEntry ← Table.Lookup(p.EthDst)
45:  if !DstEntry or IsPortDown(DstEntry.Port) then  // No valid entry.
46:    if p.L then
47:        return false  // Packet has hairpinned already. Drop and give up.
48: end if
49:
50:  p.F ← true  // About to flood the packet.
51:  if p.HC == 1 then
52:     // This is the packet’s first hop. L is already set.
53:     Flood(p)  // Flood learnably out all ports except InPort.
54: else
55:     p.L ← false  // Not the first hop, don’t learn from the flood.
57:     Flood(p)  // Sends out all ports except InPort.
59: end if
60: else if DstEntry.Port == p.InPort then  // Packet wants to hairpin.
61:     if p.L then
62:         p.L ← false  // If learnable, try once to send it back.
63:         Output(p, p.InPort)
64:     end if
65: else
66:     Output(p, DstEntry.Port)  // Output in the common case.
67: end if
```
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, rather than sending at a fixed rate,
we create flows in order to maintain an average rate (choosing
flow sizes from an empirical distribution).

We generate traffic somewhat differently for the two scenar-
ios. For the datacenter 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 receivers
(each set equaling one quarter 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.

In terms of UDP sending rates, in the datacenter case we
use a per-host rate of 100 Mbps, and evaluate using both 10
Gbps and 1 Gbps links. In the campus case, we use a per-host
rate of 1 Mbps, and again we test both 10 Gbps links and 1
Gbps links. For TCP, we pick the arrival patterns to roughly
match these per-host sending rates.

In terms of failures, we perform two classes of simulations:
one using no link failures (for comparison), and one using a
randomized failure model based on the “Individual Link Fail-
ures” in [14] but scaled to a considerably higher failure rate
in order to better demonstrate results under failure conditions
for simulations of manageable duration.

### 3.2 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 datacenter 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).

### 3.3 Dynamic Networks

To characterize the behavior of AXE in a network undergoing
failures and recovery, we first look at the number of dropped
packets with UDP traffic, which is shown in Figure 2. These
conditions represent a very high failure rate: 15 failures over
one minute for the datacenter case and 171 failures over one
minute for the campus case. In the datacenter case, AXE
incurs zero drops, 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 no unnecessary losses,
while Idealized Routing has significantly more.

TCP recovers losses through retransmissions, so we instead
measure the impact of routing on flow completion time (FCT).
We find that when comparing FCTs under AXE and Idealized
Routing, either they are very close, or Idealized Routing is
significantly worse (by two seconds or more) due to TCP
timeouts. Figure 3 shows the number of flows where the flow
completion times using Idealized Routing are significantly
worse than when using AXE (there are no cases where AXE
is significantly worse than Idealized Routing).

Lastly, we look at whether AXE’s use of flooding upon
failure imposes too heavy a burden on the network. For this
metric, we examine all packets seen on every link, and find
the fraction that are being flooded (i.e., the ones with the F
bit set). Figure 4 shows that the traffic devoted to floods is quite
small, even under our extremely stressful failure scenarios.

### 3.4 Wilt Filter Size

Deduplication using the wilt filter method is subject to false
negatives – it may sometimes fail to detect a duplicate. When
this happens occasionally, it presents little problem: dupli-
cates are generally detected on neighboring switches, at the
same switch the next time it cycles around, or – in the worst
case – they reach the maximum hop count and are dropped.
However, persistent failure to detect duplicates runs the risk

---

Note that this is the fraction of offered traffic, not the fraction of
the link, that is consumed by floods.
of creating a positive feedback loop: the failure to detect dupli-
cates leads to more packets, which further decreases the
chance of detecting duplicates.

The false negative rate of the wilt filter is inversely corre-
lated 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 simul-
ations were a worst case, as we used the UDP traffic model
(which, unlike TCP, does not back down when the network
efficiency begins degrading), and we did not use the global
flood quota described in Section 2.1.

Figure 5 shows the numbers of lost packets (which we use
as evidence of harm caused by false negatives) for the data-
center network with 1 Gbit links. Even under heavy failures,
the number of losses goes to zero with very modest sized
filters (≈500). Unsurprisingly, the number required to achieve
lossless performance with 10 Gbit links was even smaller.

3.5 Summary

Our simulations indicate that AXE works — using links as
effectively as shortest path routing (and ECMP) and recov-
ering from failures rapidly while supporting plug-and-play
(as we installed no routing state ahead of time). The biggest
remaining question is how well it scales as the number of
hosts grows. Preliminary simulations on our campus network
with 10,000 hosts indicate that the percentage of flood pack-
ets remains low (0.71%) and there are no unnecessary packet
drops even in unrealistically severe failure scenarios, all with
a relatively small wilt filter. We expect that AXE scales to
even larger sizes, and are actively exploring its scaling limits.

4 Discussion and Future Work

What we have presented here is a very preliminary version
of what we hope is a promising approach. There are many
other design options to be explored, and they fall in to two
categories: improving current features or adding new ones.

In terms of improving the implementation of features al-
ready present in AXE, we continue to look at alternate ways
of: preventing loops (using timing-based learning to prevent
loops), detecting duplicates (using a sliding window to track
sequential nonces per source), failure response (sending pack-
et back to their source before reflooding), route optimization
(by having periodic floods, so that AXE does not have persis-
tent suboptimal routes), and meltdown prevention (pruning
unicast addresses when hosts are not responding). All of
these will be more fully explored in future work.

More interestingly, there are ways we can expand the func-
tionalities of AXE. For instance, a trivial change allows AXE
to mimic the per-VLAN functionality of many STP variants
e.g., PVST, PVST+, MSTP, MISTP), although AXE’s ability
to use all links and recover from failures quickly may render
some of the motivation for this moot. More interestingly, we
are currently evaluating an AXE-native multicast design with
fast recovery properties similar to our unicast design, as well
as a preliminary anycast design. We are also pursuing a hybrid
which layers Hedera-like traffic engineering atop AXE,
allowing AXE to handle mice flows and recovery quickly
while the TE solution handles elephants efficiently.

Beyond improving and extending AXE, hardware imple-
mentation is another path for future work. An exciting first
step we are pursuing is a P4 version, and we note that
the basic algorithmic pieces of AXE (most significantly, the
packet deduplication) are implementable within P4 with a
small caveat. As there is no P4 action to add new table rows,
whenever a new MAC is first observed, an agent (running on
the switch) is required to add corresponding new table entries.

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 shortest paths, but also ECMP-
like behavior), and direct multicast and anycast support —
while retaining Ethernet’s plug-and-play characteristics. 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.

5 Acknowledgements

This material is based upon work supported by sponsors in-
cluding Intel, AT&T, and the National Science Foundation
under Grant No. 1117161, 1343947, and 1040838.

This is a problem that all learning based solutions face: if host
A never sends a packet, then all packets sent to it will always be
flooded.
6 References


