2010 International Conference on Distributed Computing Systems

'Ethernet on AIR' : Scalable Routing in Very Large Ethernet-based Networks

Dhananjay Sampath[*], Suchit Agarwal[*]
[*]Computer Engineering Department
University of California, Santa Cruz
Santa Cruz, CA 95064
Email: dsampath, agarwal@soe.ucsc.edu

J.J. Garcia-Luna-Aceves[†*]
[*]Computer Engineering Department
University of California, Santa Cruz
Santa Cruz, CA 95064
Email: jj@cse.ucsc.edu

***Abstract*—Networks based on Ethernet bridging scale poorly as bridges flood the entire network repeatedly, and several schemes have been proposed to mitigate this flooding problem; however, none have managed to eliminate flooding completely. We present Automatic Integrated Routing (AIR) as the first routing protocol that eliminates flooding by assigning prefix labels to switches and building a Distributed Hash Table (DHT). The DHT maps host identifiers to the prefix labels of the switches through which they connect to the network. Each switch is assigned a prefix label using neighbor-to-neighbor messages. Prefix labels denote the locations of the switches in the network, and the prefix labels of any two switches automatically determine one or multiple routes between them. The DHT stores the mapping between the name of a host and its network location (prefix label) in a scalable fashion, with any one switch storing only a fraction of all the mappings. In contrast, prior approaches using DHTs to resolve host names incur the communication and storage overhead introduced by an underlying link-state routing protocol. Results using packet-level traces of Internet traffic demonstrate that AIR attains performance gains of orders of magnitude over Ethernet bridging and prior DHT-based schemes.**

I. Introduction

The rapid growth of enterprises in the last decade created a large demand for networks that are easy to setup, economical and fast. This has resulted in the emergence of Ethernet as a viable option that worked out-of-the-box. However, with several enterprises further expanding in scale and with a steep increase in the volume of devices connected to the network, limitations of Ethernet-like approaches have become apparent.

The simplicity of Ethernet, as it turns out, is its Achilles heel. Hosts connected via Ethernet use broadcast services for point-to-point communication. Bridges connecting different Ethernet segments of the network, flood an entire segment to locate destinations. Moreover, each host maintains paths to all hosts within its segment, while a bridge stores host information for all segments that it connects. This does not bode well for very large networks, because the message complexity and the local state increase exponentially with the size of the network. Broadcast-storms and routing loops arising out of transient hosts further worsen this situation.

Our work finds its motivation in that the traditional model of hosts has undergone a significant amount of change in the last decade. With advances in embedded technology, several different class of network-enabled devices have emerged and are deployed to save cost, effort and time. Each of these devices connect to the network similar to a personal laptop or a content server for the enterprise. Increasing number of hosts have so far been met with increased deployment of scale-limited networking devices and hence degrading peak performances. Our approach attempts to scale by reducing the state stored at each node, minimizing network wide broadcasts and hence improving on end-to-end latencies.

As Section II discusses, several approaches to mitigate these problems have been proposed in the literature. A careful review of them reveals that, while these approaches aim at limiting network-wide flooding, they do not get rid of it completely. Protocols based on construction of *spanning trees* help reduce broadcast storms and provide routes that are eventually loop-free. However, bottlenecks caused by the tree-like structure superimposed on the physical topology results in poor network performance. IP routing emerged as a scalable alternative, and is still a popular choice in combination with switched Ethernet. However, bootstrapping and maintaining an IP network has high management overhead, especially when the network includes transient hosts. Another popular approach involves leveraging link-state routing to set up point-to-point virtual links between hosts.

Until recently, improvements in routing for wired networks largely focussed on more efficient spanning-tree protocols. In contrast, SEATTLE [4] took a radical approach in moving away from the traditional 'broadcast-everything' approach and introduced Distributed Hash Tables (DHT) into the mix. In SEATTLE, hosts are abstracted from the rest of the network by the layer-two switches and each of these switches store the mapping between an end-host, and the switch that the end-host is connected to. However, each of these switches run link-state routing protocols on their interfaces that connect them to other switches.

In this paper, we embrace the design decision of using DHTs to resolve the mapping between host identifiers and their locations, and introduce a new approach that moves away from the need to use link-state information. Our framework pushes the responsibility of searching for destinations to nodes in the network in such a way that neither the network nor any single node is burdened with the task of discovering routes. As a result, we reduce the state stored at each node and improve

1063-6927/10 $26.00 © 2010 IEEE
DOI 10.1109/ICDCS.2010.82

IEEE computer society

the latency compared to Ethernet, without flooding messages network-wide. We call this framework Automatic Incremental Routing (AIR), which is described in detail in Section III-A.

AIR utilizes the concept of DHTs to organize each of its node as part of a routing substrate that allows for easy discovery of destinations and/or services. The way our DHT is realized in the network is different from previous approaches in that the labeling scheme supplies each node (host or a switch) with a *prefix label*. The labeled routing substrate makes it trivial to route between any two nodes with prefix labels and the discovery of a prefix label becomes analogous to the path discovery problem. To discover a path to some destination, a host resolves a well known name of the destination host to a prefix label and then routes to it. The prefix label of each host in the network is stored across the DHT and a single lookup in the DHT reveals the most recent mapping of a host's name (a well known identifier MAC or IP) to its prefix label in the network. With this information, each node proceeds to route to the discovered destination.

Section IV discusses the performance of AIR using packet-level trace files on real world topologies, as well as high-fidelity network simulations [14]. AIR achieves order magnitude gains in the reducing the control overhead and local state incurred in bridged Ethernet and SEATTLE. The dramatic performance improvements attained with AIR are due to the publish/subscribe paradigm over prefix labels enforced in AIR, which removes the responsibility of discovering paths to destinations from both the hosts and switches.

II. Related Work

Ethernet bridging, though a very flexible and rapid-to-deploy solution for local area networks (LANs), does not typically scale beyond a few thousand hosts. Every switch in an ethernet bridged network maintains routing state for all hosts in the network. Thus, the forwarding tables at switches grow very large for enterprise-level networks. Dissemination of this host information further worsens the problem. These approaches can largely be classified into three categories; (a) Schemes that augment spanning tree protocols, (b) Schemes that leverage link state based dissemination, (c) Hybrid approaches. Our approach finds its motivation from all of these works but offers a unique point in the design space in that it combines the advantages of these models while working around their drawbacks.

Spanning Tree protocols were adopted to eliminate routing loops and hence prevent repeated flooding of packets to end hosts. Rapid Spanning Tree Protocol (RTSP) [7], an augmented STP protocol, was designed to converge faster after link failures. However, STPs fail to offer path diversity and hence fail to maximize the utilization of available resources. SmartBridges [12] build multiple spanning trees and maintain complete network topology at each switch to ensure shortest path routing between segments, which does not scale well.

RBridges [9] use a link-state protocol at each bridge to learn about every other bridge in the topology. RBridges limit flooding of packets using hop counts while CMU-Ethernet [6] employs a similar link-state advertisement approach, but also includes host information in the link-state updates to avoid disseminating host location information.

Link-state routing protocols such as OSPF [5], belong to the proactive class of routing protocols where neighbors exchange link vectors between each node and build a snapshot of the entire topology locally at each node. Link updates are sent by each node to the rest of the nodes in its segment periodically, as spanning trees are computed using the local copy of the topology. Large segments running on link-state suffer from processing a high volume of updates even when no node has active traffic to the remote segments.

Subnets interconnected to each other using IP routing is used to split large topologies into smaller logical ones. The switches act as *gateways* for hosts in each subnet to communicate with other hosts. Subnets are assigned an IP prefix and hosts in the subnet are assigned an IP address from that prefix. However, the assignment of IP prefixes to subnets is typically a manual process. Although the assignment of correct IP address to hosts can be automated using DHCP services, configuring DHCP servers to allocate addresses that are consistent with subnet addresses is non-trivial.

VLANs emerged as a solution to split the network into logical groups of hosts since a single VLAN can be made to span multiple bridges. While VLANs allowed seamless host mobility as hosts retained their IP addresses, they suffered from the same problems as subnetting as manual configuration was required.

More recently, SEATTLE [4], introduced a different approach by building DHTs for discovery of end hosts. In SEATTLE, each switch did not maintain state for every host in the network and avoided network-wide flooding of announcements. Further, by piggybacking location information on ARP [10] packets and by combining caching of replies to location queries, they ensure low latency for look ups. However, the DHT is constructed on top of a link-state routing protocol and they incur all the overheads associated with that. Our work relates closest to this body in that we apply a DHT to resolve location-identity queries but is different in that we build our DHT using prefix labels and yet do not incur overheads of a link state approach. There are other works [11] that use a DHT on Ethernet to support addressing, but they assume complete knowledge of the switch-level topology, which would require either flooding or a link-state protocol on switches.

The use of a link-state protocol in SEATTLE for maintaining the switch-level topology incurs a large amount of overhead as the number of switches grow combined with increasing host and switch mobility. Thus, even though SEATTLE reduces the total control overhead it does not get rid of flooding completely. We believe there is still more room in the design space for large local area networks, where switches should not have to resort to any form of flooding of information, while still maintaining efficient and reasonably fast routing. AIR strives to fill this gap by providing automatic routing wherein switches do not need to maintain a switch-level topology in order to route between switches. The routing between switches is done automatically using the prefix labels assigned to the switches as explained in the next section.

III. AIR

Automatic Incremental Routing (AIR) is designed such that it takes the broadcasting service of the Ethernet and converts it into a virtual point to point unicast service. One of the main goals of this paper is to explore how a DHT substrate built over the switch topology can be leveraged to route to hosts. We also show that the amount of state at each *AIR-switch*[1]is a fraction of the total number of mappings and the messaging complexity is bounded to a polylog of the number of switches in the network. To the best of our knowledge this is one of the first schemes that offers succinct routing state and limited control plane signaling.

In our effort to design AIR we ensure that it can run on top of an existing Ethernet infrastructure and encapsulate bootstrapping and host-learning protocols such as DHCP [2] and ARP [10] into a single messaging primitive. Hosts attach themselves to AIR-switches, as they do in Ethernet using a *hello*-message protocol. Using the hello-messaging, AIR-switches abstract away the hosts to the rest of the network and route packets on behalf of attached hosts.

A. Routing on AIR - Overview

The AIR routing protocol routes packets across switches in three separate phases. First, in the *substrate building* phase, the protocol constructs a routing substrate in the form of a Directed Acyclic Graph (DAG) that is rooted at a distributively elected node in the network. Each node is then assigned a prefix label with respect to the root of the DAG. These nodes also have a position-independent identifier (IP or a MAC address) which we call *Globally Unique Identifier* (GID). Note that while AIR builds a DAG, it is different from the Spanning-Tree protocols in that it does not use the DAG to flood link-vectors to the entire network. The DAG merely is used to establish an ordering among the nodes with respect to the distributively elected root.

In the second phase, each switch labels every other switch to which it is physically connected. The labeling process instantiates the *hello* message protocol that sends update messages on all the interfaces of the switch and the prefix labels are piggy backed on these messages. Additionally, these messages carry a monotonically increasing sequence number that helps determine how recent an update message is and hence prevents the occurrence of routing loops. The election algorithm determines a dominant node in the connected component by choosing the node with the largest node degree and breaks any ties using GID.

Once the prefix labels are established, the *anchor setup* phase begins. In this phase, each switch in the DAG identifies an *anchor* which is responsible for storing the mapping between the GID of the host and associated prefix label of the switch. The 'anchor switch' is determined by using a consistent hashing function [3] that takes as input, the GID of a host and returns a *switch-prefix-label*. As hosts fail, reboot or even move around, their labels change and anchors keep track of these changes.

[1]Henceforth, switches and AIR-switches are used interchangeably

Note that the prefix labels are not switches but locations in the network. As the topology of the network changes over time, while the switches associated with locations in the network changes, the location themselves remain invariant, provided the labels are consistent and the network is not partitioned. The labeled DAG offers a resilient structure, that defaults to a switch with the closest switch-prefix-label, when a request for an exact match fails.

Lastly, in the *destination discovery* phase, switches requesting a route to a particular destination host, compute the anchor switch for that destination. The switch then sends a request for the most recent mapping between the GID of the destination and a switch-prefix-label. These messages take advantage of the routing substrate and route implicitly to the anchor switches. Upon reception of the mapping, data is routed either directly, using knowledge of better paths to the destination or is routed via the anchor node. In this paper, we chose the latter design to bound path lengths and quantify the worst case behavior of the protocol.

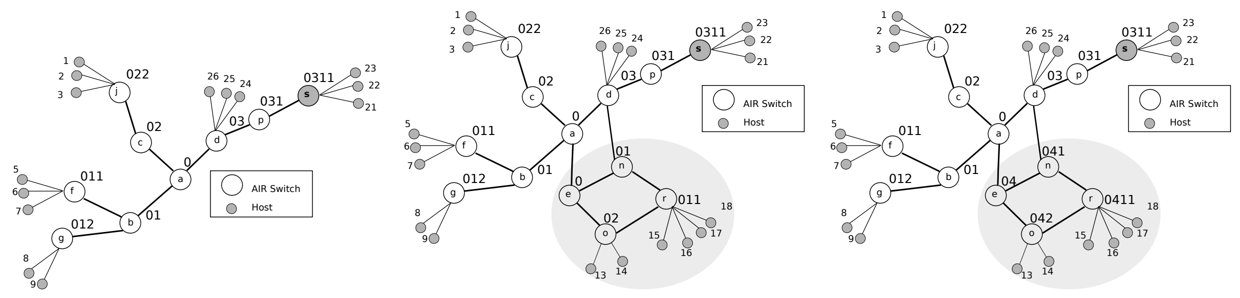
The routing algorithm that routes over the labeled DAG, follows a greedy strategy. When it encounters a local minima, it chooses the next hop with the lowest GID. Since the labels are combinations of prefixes, the choice of GID does not affect the path to the destination. A switch uses the *maximum matching prefix* logic to choose its next hop. To route to a destination d, switch $s \in E$ chooses the link that offers the maximum length of prefix label that matches with the prefix label of the destination.

B. Building the DAG

Switches in AIR are labeled such that the labels preserve an ordering among them with respect to a single elected switch. Each switch periodically evaluates its neighborhood information and determines if a neighboring switch can offer a better ordering with respect to the elected root node. If a switch detects no ordering in its neighborhood, it elects itself as a root and labels its neighbors. The ordering is determined by the lexicographic length of a switch's label and the GIDs of the switches are used to break ties, should a tie occur.

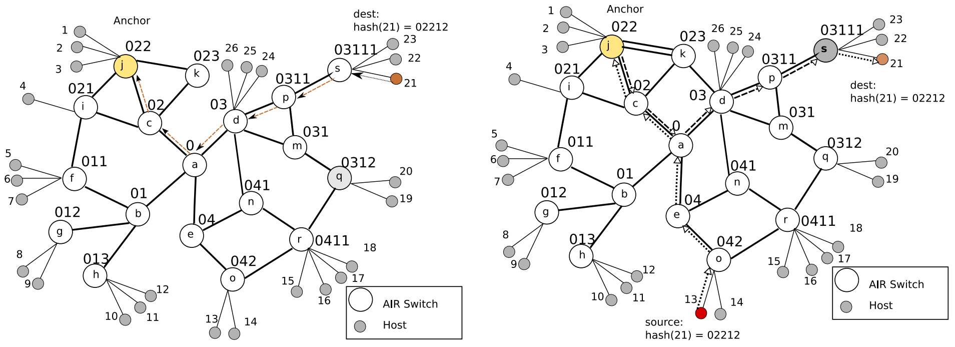
To understand the prefix labeling better, consider the example in Figure II. The prefix labels for switches are assigned over the alphabet $\Sigma = 1, 2, 3$. Switch a elects itself as the root node and labels switches b, c, d with labels $01, 02, 03$ respectively. The labels are assigned such that it is prefixed with the label of the parent switch combined with a suffix that is unique at the parent node. Here we see that switches b, c, d were given suffixes $1, 2, 3$ at switch a.

Note that as the network begins to organize itself, the labeling process can produce multiple such DAGs. When a switch is at the interface of two such DAGs, the label of the border switches are compared and the DAG with the lexicographically larger label dominates the ordering of the



(a) prefix labeling (b) a, e are at the interface of 2 labeled DAGs (c) DAG with switch a dominates

Fig. 1. Building the DAG



(a) Switch s inserting mapping into the DHT for 21 (b) Source host 13 locating host 21 via anchor j

Fig. 2. Overview of the AIR switch

switches. For example, if a switch with label 0 were to join the DAG in the Figure 1(b) at switch a then the label of node a would dominate the ordering of the new DAG joining at node a.

All nodes in the network (host or switch) auto-configure themselves with their labels using neighbor-to-neighbor signaling. Messages from hosts are handled differently from that of switches in that the messages from hosts are abstracted away by the switch. Each host sends a periodic hello-message to the switch that it is connected to and the soft state maintained at the switch determines the existence of the hosts.

Switches maintain a list of the hosts that are connected to it. In addition to this each host also sends a monotonically increasing sequence number, which is used to ensure loop-freedom when hosts travel between switches.

C. Building the DHT

'Anchor switches' store the *(key,value)* pairs that maps the switch-prefix-label with the GID of an end host. We choose consistent hashing to avoid remapping of hosts to switches when parts of the network fail. These hash functions take the GID as input and return a switch-prefix-label. Each host in the network announces its presence to a switch in its segment and provides the switch with its GID. Switches, then hash the value of the GID and *insert* the $(host_{GID}, switch^{host}_{GID})$ at the switch-prefix-label returned by hash function.

A source host with traffic for some destination learns of the destination's location from the anchor switch storing the *(key,value)* pair. The source then sends a *look-up* query towards the switch closest to the switch-prefix-label. Consider the example in Figure 2(a). Host 21 is connected to switch s and s hashes the identifier of 21 and acquires a switch-prefix-label of 0221. This mapping is then stored at switch j which is the closest match to the acquired switch-prefix-label. Note that the node with the exact label matching the switch-prefix-label does not have be present in the network. Failures or node transience are handled similarly as the DHT *inserts* add entries into a switch that matches closest to the absent switch.

D. Routing between Hosts and Switches

Once the DAG is setup by hello-messaging between the switches, packets are routed using prefix labels. The construction of the DAG forces an ordering among the switches in the network. At each hop, a switch determines the next 'best' hop towards the destination prefix label. The switch that offers the largest common matching prefix is then selected as the next best hop. DHT inserts and look-ups are propagated in this fashion to build the DHT.

Sources that have active traffic to send to destination hosts, determine the anchor switch's prefix label and send a look-up. Successful look-ups propagate back to the source node following which a source sends data packets directly to the destination without having to route via the anchor switch.

In Figure 2(b), we see that host 13 attempts to route to host 21 by first hashing host 21's identifier to get the switch-prefix-label of host j and then sends a request to host j for host 21's

Algorithm 1 Root Election

```
if 1-hop neighborhood does not have a valid parent then
  if Label Timer expires then
    Elect self as root node
    if number of neighbors is not 0 then
      assign each neighbor a unique suffix
      send Hello message with the assigned prefix labels
    end if
  end if
else
  if 1-hop neighborhood has a valid parent then
    if lexicographic length of parent label < current label
    then
      if node-id of self is less than the node-id of parent
      then
        Elect self as root and send Hello message
      else
        Wait for Hello message from parent
        Change label to prefix label provided in the Hello
        message
      end if
    else
      Wait for Hello message from parent
      Change label to prefix label provided in the Hello
      message
    end if
  end if
end if
```

prefix label. Host j in turn provides switch o (connected to host 13) with the prefix label of switch s.

Several optimizations can be envisioned to improve different metrics. To improve latency, data packets are be piggy backed with the requests, or to ensure reliable packet delivery, packets are routed through the anchor switch at all times. This is so that if a host were to relocate to another switch, the anchor would be updated with the switch to which the host is currently connected. In our implementation, we do both of the above and further to evaluate the worst case behavior of the protocol under high traffic loads and also to demonstrate the possibility of enforcing routing policies, we decided to route packets through the anchor.

Therefore, in the example, we observe that the piggybacked data packet with the request is routed from switch j to switch a (which is the least common ancestor for switches j and s) and the packet is then routed towards switch s.

1) Routing Optimizations: Optimizing routing paths comes at the cost of increased local state. To optimize the routing through the anchor switches, each node maintains additional state upto 2-hops of the neighborhood. This enable each switch to discover shorter paths to disjoint labels. Note that unlike spanning-tree protocols, shortcut paths to destination hosts are allowed. It is intuitive at this point at increasing local state and ensuring that the local state stays updated creates the trade-off between path-stretch and routing table size. While optimizations are clearly possible, in this paper we focus on demonstrating a simple design that can scale well in the number of nodes and upto a certain amount of traffic.

E. Unicast and Multicast Traffic

AIR supports both unicast and multicast traffic. In the case of multicasting, establishing receiver-initiated multicast trees is very similar to the manner in which unicast routes are established. A *group-prefix-label* is derived from a multicast group identifier and this serves as the *anchor* of the multicast group, which serves the traditional role of the core of a multicast group. To join a multicast group, a multicast receiver simply hashes the group identifier and obtains the group-prefix-label. A join-request is sent towards the node serving as the anchor of the group. A reverse path is activated as the join request is forwarded and relaying nodes become a part of the shared multicast tree for the group.

A multicast source simply sends its multicast data packets towards the anchor of the target multicast group. These packets are multicast over the shared multicast tree after they reach the first node that is already a member of the group. The overhead of constructing multicast trees is eliminated as the underlying prefix labeled DAG organizes switches as trees already and allows automatic routing using prefixes.

F. Aggregation of source traffic

To reduce repeated retransmission of different messages, we implemented a system of adaptive timers that sent a message after holding down transmissions for a certain interval. Note that this is not done for all class of traffic. Once the transmission timeout is triggered, packets are created by draining different local queues and determining the best next hops for the transmission of each message.

Messages from different hosts are aggregated and sent out as a single message. Additionally, messages traveling towards similar prefixes are grouped together and require fewer messages.

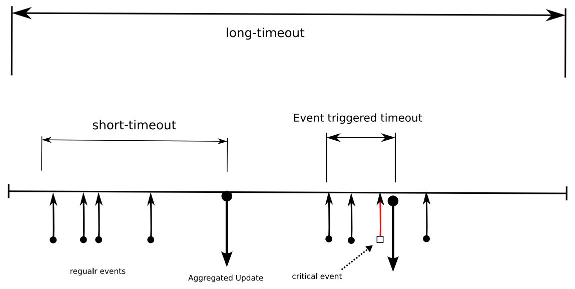


Fig. 3. Adaptive update timer and Aggregation of host traffic

G. Adaptive update timers

The firing of local update timers triggers a DHT insert. The timers for this event were typically set to twice the hold down time of a node. Upon expiry of this timer, the locality of the node is compared with the snapshot from the start of the hold down time and if no event that changes the node label occurs during this period, then the update message is clamped and delayed till the end of the next cycle of timeouts.

The DHT entries across the network are evicted based on a hard state. Freshness of updates is determined with the help of a sequence numbering scheme and time stamp of packets. This ensures that the hard state does not suffer from wrap-around effect of the sequence number.

H. Handling Host Dynamics and Switch Failures

To minimize disruptions to the DAG due to topological changes we first describe how the substrate offers intrinsic tolerance to changes. We then show how stronger disruptions are handled in a systematic fashion with the goal of preventing a network-wide reset of labels. We resort to a network-wide reset only when both these schemes fail. From observing the traffic and topology of real world traces such scenarios are atypical and account for less than 1% of the changes.

A key point in understanding the fault-tolerant nature of the design is that, in selecting the underlying structure as a DAG, we allow existence of multiple paths between nodes within a single component. This increases the number of back up paths between any two nodes in the network.

1) Anchor resilience: Assume that some switch with label $k\alpha x$ is the designated anchor for host y in the network. Host y pro-actively updates its anchor with the label of the switch that its currently connected to. If switch $k\alpha x$ fails or is relocated to different part of the network, or if no switch with such a label exists, then a switch in some subtree with the maximum matching prefix is designated as the *anchor*. Note that this anchor is logically in the path towards the original anchor. Should the topology change, and newer switches acquire labels that match the anchoring label better, then these new switches become the anchor of some hosts in the network.

To exemplify, if a switch with label $k\alpha x$ were to go down and the prefix of this label were $k\alpha$, then the last switch actively engaged in providing directory service is in the subtree α. In the worst case scenario this scheme backs up to the root switch, since no other switch has a closer prefix to the destination currently being sought.

2) Substrate resilience: If an internal switch were to fail or relocate, the subtree rooted at that switch does not reset its labels immediately. Each child of the root of a subtree determines if other peers can be reached through other paths. In this event, the peers masquerade a position in the work. This is akin to the auto-configuration of default gateways, with no need for manually specifying multiple default gateways should one fail.

IV. Performance Evaluation

In this section, we evaluate our framework in two different ways. First, we use our packet-level simulator to replay traces from the Lawrence Berkeley National Lab [8] campus network on a real-world topology. Second, we used a hi-fidelity event-driven packet level network simulator, QualNet-v.4.0 [14]. We use these two environments to showcase two aspects of our framework. The packet-level simulator of the traces allows us to scale up 60,000 nodes and helps us analyze protocol interactions in a larger-scale. QualNet simulations on the other hand gives us a more fine grained detail of routing state at each node and the interactions between data and control.

Algorithm 2 Node Dynamics

```
if 1-hop neighborhood has changed then
  if next hop to root node does not exist then
    compute changes in the 2-hop neighbors
    if valid paths exist to peer of the next hop to root node
    then
      assign self as parent of the subtree
    else
      if timer for relabeling expires then
        if next hop to root node has changed then
          initiate Relabeling logic
        end if
      else
        wait for Timeout $and$ Check for changes in 1-hop
        neighborhood
      end if
    end if
  else
    decrement child counter
    check if self is a core or an anchor for child
  end if
else
  wait for topology changes
end if
```

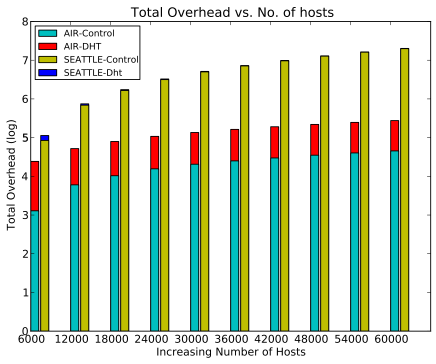
A. Trace based Evaluation : Setup

We closely followed the simulation setup described in [4] and obtained fourteen sets of traces, each lasting almost an hour. A total of 10000 hosts (sources, destinations) were described in these traffic traces. We further extrapolated the traffic sources to fit larger topologies by maintaining degree distributions of flows across the network. We calculate the control overhead for both AIR and SEATTLE over roughly 70 million end-to-end sessions. To instrument a large scale topology we chose AS 1239 topology [13] with roughly 700 routers.

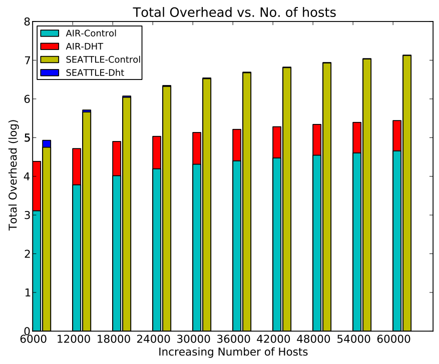
In our simulations for the large topology we assume that hosts are connected to switches and are distributed uniformly across these switches. We chose this instead, since our interest was in the behavior of the switches once the hosts are abstracted away.

We benchmark AIR with SEATTLE in these sets of experiments. We run AIR on the switches, even though hosts can also run the same protocol with trivial modifications. SEATTLE is run on the switches as well and while we take into account the *register* messages sent by the host, we do not count the ARP or DHCP requests send out by these hosts.

In addition to simulating the application level traffic and correspondingly the overhead incurred from those sessions, we also simulate the underlying link state routing protocol in the



(a) Link State Time Out 1 minute



(b) Link State Time Out 30 minutes

Fig. 4. Packet level simulation of Real traffic traces (Link State+DHT)

case of SEATTLE. We modeled the link state routing protocol similar to OLSR [1] in which the link state updates are sent at regular periodic intervals. Each of these simulations were run over 10 random seeds to avoid bias arising as an artifact out of particular topologies.

Both AIR and SEATTLE have an initial setup overhead. While AIR sets up labels using hello-messages, the link state routing algorithm underneath SEATTLE converges once every switch has been added to the topology. Even though the DHT inserts, for both protocols contribute towards the overhead we plot them separately. Notice in figure 4(a) the overhead incurred by the DHT messages is a tiny fraction of the entire routing overhead. As the network size grows, the DHT overhead increases very little.

A more interesting observation is that the overhead incurred from link state routing is higher by at least 1.5 orders of magnitude. With increasing number of hosts, we can also see that the link state overhead clearly grows much faster than the signaling incurred by routing using AIR. As one might guess, this is primarily owing to repeated link state advertisements across entire network. We also re-ran the simulations for larger time out values of the link state (30 minutes), considering the topologies are fairly static. We see in figure 4(b) that there is still an order magnitude difference between these two protocols.

AIR leverages the DHT to route packets as well as perform location look-ups and this reduces the signaling between switches. For a 10 times increase in the number of hosts in the network, we notice that while SEATTLE overhead increases 2 orders in magnitude, AIR increase hardly by a factor of one. Note that we don't simulate SEATTLE with cache enabled. Caching is done typically at the DHT level and therefore contributes to a reduced overhead in the number of DHT look-ups. However, we notice from the graphs that the fraction of DHT message contribution to the overhead is still very small compared to the contribution of the link state routing.

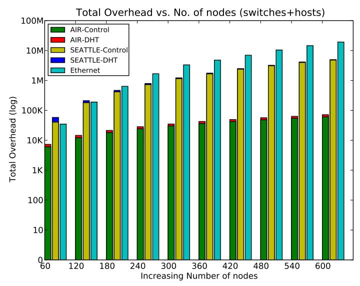
B. Simulation based Evaluation : Setup

To understand the contribution of the routing protocols to the local state, we implemented both SEATTLE and AIR in a discrete event simulator, QualNet-4.0 [14]. We ran two classes of experiments, one with increasing hosts and the other with increasing number of active flows. The first scenario characterizes the scaling properties of the protocol and demonstrates the growth rate of state and overhead with increasing number of hosts/switches in the network. The second scenario showcases the robustness of the protocol to increasing traffic demands independent of the scale of the topology, providing an insight into traffic dependent overheads.

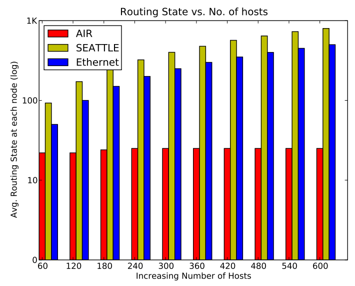
In our simulations we built a topology in which the we varied the number of switches from 10 to 100 for the increasing hosts scenario. For increasing flows, we kept the number of switches constant at 50 and varied the number of flows from 20 to 100 at intervals of 20 flows per tick. For both scenarios each switch was connected to 5 hosts on an average. Simulations were run for a duration of 2 minutes and each flow that was set up ran on an average for 1 minute. Both switches and hosts were connected over standard switched Ethernet network. The flows were setup using constant bit rate generators (CBRs). For statistical significance of our results, each simulation was run 10 times and the results are presented with a 99% confidence interval.

Switches were placed in a random graph topology with an average node degree of 5 and each interface was configured with an IP address. We used this setup to demonstrate that AIR could be run on currently existing Ethernet infrastructure. Also, in addition to SEATTLE, we benchmark our framework with Switched-Ethernet as well.

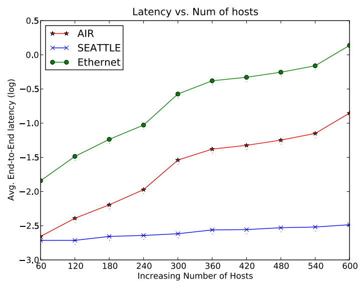
1) ***Routing State:*** AIR sets up routes between switches and each switch stores a part of the DHT. In addition to the DHT, the AIR-switches also maintain a neighbor table to keep track of the hosts/switches and the interfaces on which they are connected. In the first scenario, as the number of hosts increase, we see from Figure 5(b) that the number of table entries maintained by SEATTLE is almost 1.5 orders



(a) Control + DHT overheads incurred with increasing hosts

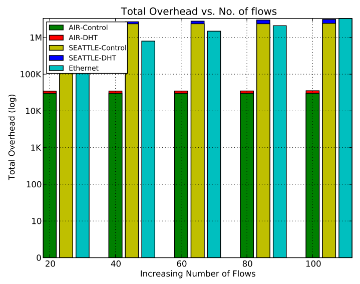


(b) Total state maintained by each node in the network (Link State+DHT)

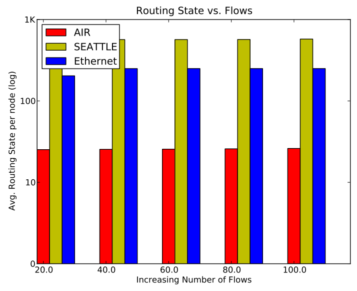


(c) Avg. E2E Latency

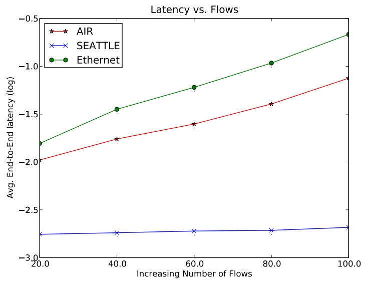
Fig. 5.



(a) Control + DHT overheads with increasing number of flows for a 200 node network



(b) Total state maintained by each node in the network (Link State+DHT)



(c) Avg. E2E Latency

Fig. 6.

of magnitude more than AIR. While we know that the DHT entries at each protocol are roughly the same, the only other contributor to the overhead is the forwarding table at each node. We notice that the routing state incurred by Ethernet is of the order of the number of nodes and SEATTLE incurs a little more owing to the DHT entries.

The largest contributor to the overhead, hence, is the link-state routing protocol. Each switch maintains an entry for every other switch in the entire network. However, since AIR leverages its own DHT messages and the prefix labels to route, it does not keep much state locally. The local state is of the order of the number of neighbors that each node has and an additional DHT overhead which is a fraction of the number of hosts (number of hosts/number of switches).

In the varying traffic scenario from figure 6(b), we notice that while a significant difference exists between the two protocols, neither of the two grows too rapidly. Reducing link state overhead however, frees up a significant amount of bandwidth in the control plane. The freed up bandwidth is in turn reallocated to the data plane for higher traffic carrying capacity. Also, in very large networks, the switches become terribly unresponsive when the number of entires in the table become large. This has adverse effects on latency and slows down look ups considerably resulting in poor network performance.

*2) **Signaling Overhead**:* The signaling overhead of AIR and SEATTLE comprises of two distinct components. The overhead from the routing protocol and the overhead from the DHT messages (inserts and lookups). Even though AIR combines the signaling messages by aggregating different types of these messages into a single packet, for the sake of presentation, we count each DHT request/look up as a separate message.

In figure 5(a), we clearly see the differences in the orders of magnitude of Ethernet, SEATTLE and AIR . The graph also shows the corresponding contribution of DHT overhead of each protocol (excluding Ethernet). The contribution of the DHT overhead to the total control overhead is remarkably small. Moreover the DHT based overhead for AIR and SEATTLE are comparable. Clearly the number of messages sent by the link-state protocol, while being an order of magnitude less than Ethernet is still worse than what can be achieved by leveraging the DHT messages for routing. While SEATTLE can be made at least an order more efficient than Ethernet by optimizing the link-state flooding in the network, AIR still manages to do better by 1.5- 2 orders.

In the scenario with increasing flows (Figure 6(a)), we don't see much of a difference between increasing flows as most control overhead is independent of the traffic. Although, one can imagine that if the network was run at capacity, gains from clearing the control-plane and consequently providing a larger data-plane can be significant. Aggregation of messages also contributes to the reduction of the number of messages in the network. We don't present the additional results owing to a lack of space.

C. *Latency*

End-to-End latency was measured as the amount of time it took for a source to send a packet to the destination, averaged over all flows during the course of the simulation. We observe from Figure 5(c) that while AIR performs better than Ethernet, it is slower than SEATTLE by at least 1.5 orders of magnitude. A primary reason for this is that while the link-state routing protocol converges to shortest path routes, routes computed by AIR have a routing stretch greater than 1.

Additionally, as mentioned before, we selected a simple routing strategy that involved routing all data packets via the switch-anchor. Since we continue forwarding traffic through the switch-anchors post-resolution of the destination's host address, data packets continue to travel over longer paths. Additional state at each node (to include 2-hop) neighbor information can alleviate this problem by exploiting shortcuts in the DAG to route to destinations. We look into this as a part of our ongoing investigation into the development of the protocol.

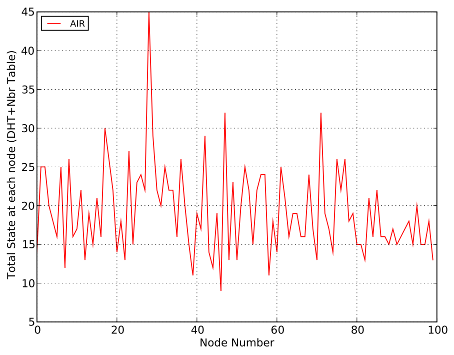


Fig. 7. Routing state at each AIR-switch; 100 switches, 500 hosts, 200 flows

V. Future work

AIR incurs an expense of a path stretch greater than one. As enterprises require guarantees on paths that the packets take it would be interesting to see if we can achieve bounded stretch. In this paper, we do the worst case analysis by routing packets through anchors, but still see a drop in latency as compared to Ethernet. Solutions to overcome non-optimal stretch has typically involved increasing the routing state.

In Figure 7, we observe that the distribution of the key space is not uniform. However, notice that most of the nodes have total state that lies within the standard deviation and all of them lie within 2*standard deviation. This means that a few nodes store more information than the others but the number of entries that they store is not a lot and decreases with increasing number of nodes.

VI. Conclusions

In this paper, we presented a routing framework, that builds and leverages DHTs in the Ethernet over prefix labels. While a complete solution that optimizes space, message complexity and time still escapes us, we strongly believe that DHTs help reduce overhead in large-scale wired networks. AIR mitigates the problems of scaling and efficiency in traditional Ethernet, while getting rid of the complexities of managing an IP network. This is the first approach, to our knowledge, for routing in local area networks that completely eliminates flooding of information for host-location resolution. We also show that by doing away with link state we can still route packets between end hosts. Our experiments show that AIR routes with 3 orders of magnitude of control overhead lesser than Ethernet or SEATTLE and scales better than most previous approaches.

References

[1] T. Clausen and P. Jacquet. *Optimized Link State Routing Protocol (OLSR)*. RFC Editor, United States, 2003.
[2] R. Droms. Dynamic Host Configuration Protocol. RFC 2131 (Draft Standard), Mar. 1997. Updated by RFCs 3396, 4361, 5494.
[3] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, 1997.
[4] C. Kim, M. Caesar, and J. Rexford. Floodless in seattle: a scalable ethernet architecture for large enterprises. In *SIGCOMM '08: Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, 2008.
[5] J. Moy. *OSPF Version 2*. RFC Editor, United States, 1998.
[6] A. Myers, E. Ng, and H. Zhang. Rethinking the service model: Scaling ethernet to a million nodes. In *ACM SIGCOMM HotNets*, 2004.
[7] R. Pallos, J. Farkas, I. Moldovan, and C. Lukovszki. Performance of rapid spanning tree protocol in access and metro networks. In *Access Networks & Workshops, 2007. AccessNets '07. Second International Conference on*, pages 1–8, Aug. 2007.
[8] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney. A first look at modern enterprise traffic. In *IMC '05: Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*.
[9] R. J. Perlman. Rbridges: Transparent routing. In *INFOCOM*, 2004.
[10] D. Plummer. Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware. RFC 826 (Standard), Nov. 1982. Updated by RFCs 5227, 5494.
[11] S. Ray, R. Guerin, and R. Sofia. A distributed hash table based address resolution scheme for large-scale ethernet networks. In *Communications, 2007. ICC '07. IEEE International Conference on*, pages 6446–6453, June 2007.
[12] T. L. Rodeheffer, C. A. Thekkath, and D. C. Anderson. Smartbridge: a scalable bridge architecture. In *SIGCOMM '00: Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2000.
[13] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. Measuring isp topologies with rocketfuel. *IEEE/ACM Trans. Netw.*
[14] S. N. Technologies. Qualnet. http://www.scalable-networks.com/.