Standards-Based Discovery of Switched Ethernet Topology

Mark R. Meiss
Indiana University

Steven S. Wallace
Indiana University

Abstract

Because static network maps are difficult and expensive to maintain, the ability to discover the topology of a switched Ethernet broadcast domain is valuable. This paper describes a standards-based algorithm for constructing the topology of such a domain, demonstrates its correctness, suggests several ways that the speed of this algorithm can be increased, and comments on problems that have been observed in automatic topology discovery.

1 Motivation

In order to manage a network well, network administrators need to know the physical layout of then network. The reasons for this are myriad. Knwoledge of what equipment has been deployed already, and where, drives plans for expansion and improvement of the network. Most network problems today occur in a user's local broadcast domain, and locating the cause of a problem in a switched network is almost impossible without having some knowledge of the network topology. Topology information is also important for maintaining network security, as it is essential to the process of tracing a rogue system down to an individual data jack.

Maintaining topology information as a set of hand-maintained static maps consumes a great deal of time and energy and often suffers from poor accuracy. The information represented in static maps is a moving target, and for a large campus network it is unlikely that a complete network topology map for the campus that is manually updated will ever be completely accurate. Maintaining topology information in a relational database and using a

network management application to draw maps eliminates some of the costs associated with keeping maps updated, but it still suffers from the transitory nature of the data.

Another serious drawback of static network topology information is that it does not reflect the *operational* topology of the network. In a network that contains redundant switching connections that are managed by Spanning Tree Protocol, a static network map offers no hint as to which of several redundant connections will be in service at any given moment. Dealing with the operational topology of a network also eliminates a host of headaches familiar to network administrators who are forced to work from static diagrams: typographical errors in network addresses, mislabeled ports, undocumented equipment, and so forth.

A key advantage of operational topology information is that by its very nature, it is available online to network management applications that need it. Carefully drawn network diagrams are of no use from a programmatic standpoint; accurate topology information must be made directly accessible to network tools or it is of limited use.

Operational topology information also satisfies the need to be able to determine the switching path that is taken between the last-hop router and a host. Experience shows that most network problems today can be found in the local network segment; most of these problems can be located only if the switching path to a host is known. Some of the problems that can be discovered and resolved with knowledge of this path include media problems, duplex mismatches, and user-deployed network hubs. In addition, determination of the switching path to a host is key to enforcing network security, as the identity of the last switch port in the path can often be matched against a list of data jacks to pinpoint a definite physical location for a host.

Almost as important as the benefit of being able to determine the operational topology of a network is the requirement that the discovery technique be based on common standards. A number of network equipment vendors have devised schemes by which one can determine the operational topology of their equipment, but this is of little use in a heterogeneous netowkr containing equipment from several vendors. Manufacturers come and go and prices rise and fall; no network administrator finds it desirable to be locked into a single brand so that topology will function correctly.

The need, then, is for a method of discovering the operational topology of a switched Ethernet network in real time using only the standard SNMP Management Information Bases (MIBs). It must also be possible to use this

discovered topology to determine the exact switching path taken between the gateway router and any host in the network.

The remainder of this paper describes a technique for constructing this topology information and suggests a number of ways in which its speed and efficiency may be improved.

2 Types of Topology Discovered

The main sources of information for topology discovery are the ARP table of the gateway router and the switch forwarding tables of individual switches on the network. All routers and Ethernet switches implementing the standard SNMP MIBs will have this information available. The algorithm described in this paper will thus work on any switched Ethernet network irrespective of whether it runs at 10, 100, or 1000Mbps.

It must be noted that this algorithm determines only the *operational* topology of the network. If the design of the network includes switches with redundant connections to one another that are managed with Spanning Tree Protocol, then only the currently active links will be discovered. The Spanning Tree connections that are in stand-by mode will not appear to be part of the operational network.

Another caveat is that this algorithm is not capable of identifying hubs and repeaters in the network in a concrete manner. The reason for this is that the standard MIB for managed hubs does not include any relevant per-port state information beyond the last MAC address which originated traffic on the port. This information is not sufficient for reliable construction of network topology. On the other hand, the constructed topology is likely to give strong hints as to where in the network hubs may be present. Hubs will appear as places where a network cable seems to have more than two ends; if any switch port is connected to more than one other device, there is likely to be a hub present.

Finally, it goes almost without saying that all switches in the network must support management via SNMP and have a network management adress within the broadcast domain of the network, though we do not need a list of switches present in the network before we apply the algorithm.

3 The Topology Discovery Algorithm

3.1 Notational Conventions

Before we describe the topology discovery algorithm itself, it is useful to define some notational conventions.

- Let $\mathbf{N}$ be a subnet in the Internet Protocol (IP) address. An example of an IPv4 subnet expressed using the CIDR notation would be 10.150.52.0/24. It is, of course, possible for a switched Ethernet network to have more than one subnet present, but it will simplify the algorithm to deal with the case of a single subnet. Extension of this algorithm to work across multiple subnets is straightforward.

- Let $N \in \mathbf{N}$ denote an IP address N that is a part of subnet $\mathbf{N}$.

- Let $\mathbf{M}$ denote the set of all Ethernet MAC address and let $M \in \mathbf{M}$ denote a specific MAC address.

- Let N_B denote the IP broadcast address within the subnet $\mathbf{N}$. All packets with N_B as a destination address are sent to every host in N. This address is equal to the subnet address of N with the host field bits set to all 1's. Thus, the broadcast address for 10.150.52.0/24 is 10.150.52.255.

- Let $H \notin \mathbf{N}$ denote the IP address of the host running the topology discovery algorithm. We restrict H from being in $\mathbf{N}$ because if H were in $\mathbf{N}$, it would not be able to force a router in $\mathbf{N}$ to accumulate ARP entries for the switches in $\mathbf{N}$.

- Let $R \in \mathbf{N}$ denote an Internet Protocol address of a router such that all traffic between H and a host in $\mathbf{N}$ will pass through R. Note that the algorithm works without modification in the event that $H = R$.

- Let $S_i \in \mathbf{N}$ denote the management IP address of a switch.

- Let $P_{S,j}$ denote the SNMP *dot1dBasePort* value of the jth port on switch S. This is a value found in the *dot1dBasePortTable* table of SWITCH-MIB and is the port enumeration used in the SNMP view of the switch forwarding table. *dot1dBasePort* for a port is not necessarily equal to *ifIndex* for a port, though this is often the

case. The $ifIndex$ of a switch port can be determined using the $dot1dBasePortIfIndex$ object.

- Let P_R denote the SNMP $ifIndex$ value of the interface on the router R associated with the IP address of R.

- Let $ARP(R, N) \mapsto M$ represent the process of using SNMP to query the ARP table on the router R for the MAC address M associated with the IP address N.

- Let $Fwd(S, M) \mapsto P_{S,j}$ represent the process of using SNMP to query the switch forwarding table on the switch S for the port $P_{S,j}$ on which there is an entry for the MAC address M in the forwarding table.

- Let $Seed(N)$ represent the process of sending a single UDP packet with an empty payload from H to N. The algorithm will perform this operation to ensure that R will issue an ARP query for the MAC address of N before we begin topology discovery. Unless this is done, a directed discovery packet to N may be dropped by R as it stops to ARP for N.

- Let $Switch?(N) \mapsto \{true, false\}$ represent the process of sending an SNMP query to N that requests the values of three standard MIB objects: $sysServices$, $ipForwarding$, and $dot1dBaseNumPorts$. The value of $Switch?(N)$ is $true$ if $sysServices$ indicates that N offers layer two (data link layer) services, $ipForwarding$ indicates that N does not do IP forwarding, and $dot1dBaseNumPorts$ has a value. The value of $Switch?(N)$ is $false$ if any of these variables have a different value or if N does not respond to the query.

- Let $\{x\} \ni \mathbf{S}$ denote the assignment $\mathbf{S} \leftarrow \mathbf{S} \cup \{x\}$.

3.2 Data Structures

We now describe the data structures that will be used by the algorithm. All of these structures are empty at the start of the discovery process unless otherwise noted.

- Let $ip : \mathbf{M} \rightarrow \mathbf{N}$ be an associative array that maps a MAC address M to its associated IP address N.

- Let $mac : \mathbf{N} \to \mathbf{M}$ be an associative array that maps an IP address N to its associated MAC address M.
- Let $uplink : \mathbf{N} \to \mathbf{P_N}$ be an associative array that maps an IP address N to the port $P_{N,j}$ on N that is closest to R.
- Let $ports : \mathbf{N} \times \mathbf{P_N} \to \wp(\mathbf{N})$ be an associative array that maps an IP address N and port $P_{N,i}$ on N to the set $\{N'_0, N'_1, \ldots, N'_k\}$ of IP addresses of switches known to be connected, directly or indirectly, to that port.
- Let T be the topology tree. Initially, T has only one node, R. The notation $T.add(S,\ P_{S,i},\ S',\ P_{S',j})$ denotes adding a new child node S' to port $P_{S,i}$ of the node S. $P_{S',j}$ is the port on S' that leads back to the root of the tree.

3.3 The Algorithm

We now describe the algorithm itself, which is presented in terms of a main routine $Main()$ which then dispatches to a recursive routine $Topology()$.

$Main(R, \mathbf{N})$:

1. For each $N \in \mathbf{N}$, do $Seed(N)$.
2. $\mathbf{L} \leftarrow \emptyset$.
   For each $N \in \mathbf{N}$, if $Switch?(N)$, then $\{N\} \ni \mathbf{L}$.
3. $ip[ARP(R, R)] \leftarrow R$.
   $mac[R] \leftarrow ARP(R, R)$.
4. For each switch $S_i \in \mathbf{L}$:
   (a) $ports[R, S_i] \leftarrow P_R$.
   (b) $ip[ARP(R, S_i)] \leftarrow S_i$.
   (c) $mac[S_i] \leftarrow ARP(R, S_i)$.
   (d) $uplink[S_i] \leftarrow Fwd(S_i, mac[R])$.
   (e) For each switch $S_j \in \mathbf{L}$:
       $\{S_j\} \ni ports[S_i, Fwd(S_i, mac[S_j])]$.
5. Call $Topology(R, \mathbf{L})$.

$Topology(C, \mathbf{B})$:

— C is the IP address of the device at the root of this substree.

— $\mathbf{B}$ is a set of switches known to be below C in the topology tree.

1. If $\mathbf{B} = \emptyset$, return.
2. For each port $P_{C,i}$ where $ports[C, P_{C,i}] \neq \emptyset$:
   (a) If $P_{C,i} = uplink[C]$, then skip this port.
   (b) $\mathbf{B}' \leftarrow \mathbf{B} \cap ports[C, P_{C,i}]$.
   (c) For each $S \in \mathbf{B}'$:
      i. For each $S' \in \mathbf{B}' - \{S\}$, if $S \in ports[S', P_{C,i}]$ and $P_{C,i} \neq uplink(S')$, then skip this address.
      ii. $T.add(C, P_{C,i}, S, uplink(S))$.
      iii. Call $Topology(S, \mathbf{B}' - \{S\})$.

3.4 Proof of Correctness

We now provide a proof that this algorithm will correctly generate T as seen from the point of view of R.

Although SNMP uses UDP as a transport protocol and is thus susceptible to lost packets, we will make the assumption that the path between H and R and between H and each switch S is of sufficient quality so that given an appropriate timeout period and number of retry attempts, we will receive a response to a valid SNMP query. We also assume that we are using the correct public SNMP community string for querying all the switches in $\mathbf{N}$.

We must now demonstrate that the information we gather in $Main()$ before dispatching to $Topology()$ will actually be available in the SNMP agents of each device. There are three main phases of SNMP queries: the identification of switches, the collection of ARP data, and the collection of switch forwarding data. We consider these three phases separately.

1. The $Switch?(N)$ function does provide reasonable identification for a switch. Every SNMP device is required to provide the $sysServices$ and $ipForwarding$ variables, and every switch with a forwarding table that we can query with SNMP will have the $dot1dBasePort$ variable present. We may exclude hybrid switch-router devices, but in the domain of topology discovery, these devices are better considered as routers than as switches.

2. R will have an ARP entry for each switch S because of the $Seed(N)$ operation. When the UDP packet directed to N reaches R, R will issue an ARP-request packet for the N if it does not already have one. In this case, R is also likely to drop the UDP packet, but that does not affect the performance of our algorithm, as we do not expect any reply from these packets.

   It is possible that the ARP entry in R for an individual switch S will expire during the query process, but this can be handled by simply directing another packet of any type toward S.

3. The matter of switch forwarding tables is more complicated. We cannot be assured that every switch in $\mathbf{N}$ will have knowledge of every switch in $\mathbf{N}$; it is quite possible that at they will not come into contact with each other's management traffic at any point during the discovery process.

   However, even though we query each switch for the port associated with every other switch, we do not need to receive all of this information. For each switch S, it is only necessary that S have an entry for R and an entry for each member of its set of descendents $\mathbf{D}$ in the topology tree.

   This information will be present. S will know of R because R has passed network traffic to S. S will know of each child $D \in \mathbf{D}$ because D has passed traffic to R through S. We query S for every other switch only because we do not know beforehand which switches are in $\mathbf{D}$ and which are not.

Thus, we are assured of being able to use SNMP to gather all necessary information for topology construct as long as the switches are following the standard algorithm for maintaining their switch forwarding tables. A problem may arise for switches that are configured to have an "uplink port" that is not included in the switch forwarding table. In the section on potential obstacles, we will discuss why this is a pathological configuration for switches and suggest heuristics for working around the problem.

At this point, we have concluded that the information stored in the associative arrays ip, mac, $uplink$, and $ports$ is complete enough so that every key queried by $Topology()$ will have an associated value. We must now demonstrate that $Topology()$ constructs an accurate topology tree.

We prove this by induction on the depth of the actual topology tree τ.

Basis. When $depth(\tau) = 0$, then τ consists of a single node N. In this case, we construct T so that N is the root of T and the topology is trivially correct.

Induction. When $depth(\tau) > 0$, then τ consists of a root node N and one or more subtrees $\{\tau_1, \tau_2, \ldots, \tau_k\}$. Assume that we construct the correct subtrees $\{T_1, T_2, \ldots, T_k\}$ and let $\{N_1, N_2, \ldots, N_k\}$ be the root nodes of those trees. The root of T will be N.

For each N_i, we know that $N_i \in ports[N, P_{C,j}]$ for some j because N_i is a descendant of N in τ. Furthermore, for every other switch $S \in ports[N, P_{C,j}]$, $N_i \in ports[S, P_{S,k}]$ only if $k = uplink[S]$. If this were not the case, then N_i could not be the root of τ_i. Therefore, we make a path from N to N_i.

τ and T are thus identical.

4 The Path Discovery Algorithm

4.1 Notational Conventions

We retain all of the notational conventions that we used in describing the topology algorithm, with the following additions:

- Let $C \in \mathbf{N}$ be an IP address of the host to which we want to determine the switching path. We require that C have a currently functioning TCP/IP stack.

- Let P_{NIC} denote the network interface card that is currently associated with the address C on the host. We introduce this convention because we do not require that C be running an SNMP agent; thus, $P_{C,i}$ may have no meaning for C.

- Let $T.root$ denote the root of the tree T.

- Let $T.children(P_{T.root,i})$ denote the set of child trees such that if $T' \in T.children(P_{T.root,i})$, then $T.root$ is directly connected to $T'.root$ on port $P_{T.root,i}$.

- Let $Ping(N)$ represent the process of sending an ICMP $echo-request$ packet to the host N.

4.2 Data Structures

We retain all of the data structures that we used in describing the topology algorithm and assume that the associative arrays *mac*, *ip*, *uplink*, and *ports* are still populated. We also assume that we have applied the topology algorithm so that we have the topology tree T for $\mathbf{N}$.

In addition, we add the following data structure:

- Let Q be a queue which will be used to hold the individual steps in the switching path from R to C. Each element in this queue is a quadruple $(N_1, P_{N_1,i}, N_2, P_{N_2,j})$, where N_1 is the source host, $P_{N_1,i}$ is the source port on N_1, N_2 is the destination host, and $P_{N_2,j}$ is the destination port on N_2.

4.3 The Algorithm

We now describe the algorithm itself, which is presented in terms of a main routine $Main()$ which then dispatches to a recursive routine $Path()$.

$Main(C)$:

1. $Seed(C)$.
2. $Ping(C)$.
3. $ip[ARP(R, C)] \leftarrow C$.
   $mac[C] \leftarrow ARP(R, C)$.
4. Call $Path(T.children(P_R), R, P_R)$.

$Path(\mathbf{B}, N, P_{N,i})$:
— $\mathbf{B}$ is the set of trees that may contain the rest of the path to C.
— N is a source host in the path awaiting a destination.
— $P_{N,i}$ is the source port on N.

1. For each tree $T_i \in \mathbf{B}$:
   (a) $P \leftarrow Fwd(T_i.root, mac[C])$.
   (b) If P is defined and $P \neq uplink[T_i.root]$, then:
      i. If $T_i.children(P) \neq \emptyset$, then $Q.enqueue(N, P_{N,i}, T_i.root, P)$.

ii. Call $Path(T_i.children(P), T_i.root, P)$ and return.

2. $Q.enqueue(N, P_{N,i}, C, P_{NIC})$.

4.4 Proof of Correctness

We now provide a proof that this algorithm will correctly generate Q as seen from the point of view of R, assuming that all existing data is correct.

We will again make the assumption that our network links are of sufficient quality so that given an appropriate timeout and number of retry attempts, we can reliably exchange SNMP packet with all managed network devices in $\mathbf{N}$. We also continue to assume that we have the correct community string for querying all managed devices in $\mathbf{N}$.

First we must demonstrate that the necessary switch forwarding entries will be present in the switches when we query them. This will be the case because of the $Ping(C)$ operation performed in the first step of the algorithm. We have already required our management station H not be in $\mathbf{N}$ and that all traffic from H to hosts in $\mathbf{N}$ pass through R. Thus, the path from R to C traversed by the ping packet will be exactly the path we wish to discover. The switch forwarding tables of each switch in the path will be initialized by the ICMP $echo - reply$ packet sent from C back to H.

Next, we will also be assured of being able to obtain ARP information from R about C because of the $Seed(C)$ operation; the same argument used in the proof of the topology algorithm applies here.

We have shown that the data we need to gather by SNMP will be available. Now we prove by strong induction on the length of the actual path κ that $Path()$ constructs an accurate switching path from R to C.

Basis. When $length(\kappa) = 1$, then the only step in κ is (R, P_R, C, P_{NIC}). We call $Path()$ with the subtrees of T, R, and P_R. For every subtree T_i of T, $Fwd(T_i.root, mac[C])$ is either not known or equal to $uplink[T_i.root]$. Thus, we will enqueue (R, P_R, C, P_{NIC}) and return. $Q = \kappa$.

Induction. When $length(\kappa) > 1$, then we know that $\kappa = (\kappa_0, \kappa')$, where $\kappa_0 = (R, P_R, S_i, Fwd(S_i, mac[C]))$ and κ' is a series of one or more steps that goes from $(S_i, Fwd(S_i, mac[C]))$ to (C, P_{NIC}), for some for some switch S_i and port $Fwd(S_i, mac[C])$ on S_i.

Because we know that T is the correct topology tree for $\mathbf{N}$, it must be that $S_i \in T.children(P_R)$. We also know that $Fwd(S_i, mac[C])$ is de-

fined and not equal to $uplink[S_i]$. Furthermore, we know that for all $S_j \in T.children(P_R), i \neq j$, $Fwd(S_j, mac[C])$ is either not defined or equal to $uplink[S_j]$. Thus, we do reach step $1.b.i.$ of the algorithm and do so only for the tree that has S_i as its root.

In that step, we see that $length(\kappa) > 1$, so the test $T_i.children(P) \neq \emptyset$ will be true, and we enqueue $Q_0 = (R, P_R, S_i, Fwd(S_i, mac[C]))$. The remainder of Q, denoted Q', is then the path from $(S_i, Fwd(S_i, mac[C]))$ to (C, P_{NIC}).

We have just shown that $Q_0 = \kappa_0$. Furthermore, by the inductive hypothesis, $Q' = \kappa'$. Thus, $Q = \kappa$.

5 Broadcast SNMP

5.1 Technical Description

Recall that we denoted the IP broadcast address within $\mathbf{N}$ as N_B. If we use $N_B \in \mathbf{N}$ as the destination address of an SNMP packet, that packet will be delivered to every device in $\mathbf{N}$. We can thus query many SNMP agents at once, as long as the form of the query is independent of the agent we are querying.

Because we have required $H \notin \mathbf{N}$, the router R must permit H to make directed broadcasts to $\mathbf{N}$ if we are to use this technique. This exception should be made only for H. R must disallow directed broadcasts in general; to do otherwise leaves N vulnerable to the "Smurf attack." Ideally, R should be configured to allow H to do directed broadcasts only to the standard SNMP port, which is UDP port 161.

5.2 Motivation

Given the complexity and security concerns of configuring R to allow SNMP broadcast from H to pass into $\mathbf{N}$, it is reasonable to question what benefit broadcast SNMP may have.

We address this concern by considering the number of packets generated by the topology discovery algorithm.

Let n be the number of host addresses in $\mathbf{N}$ and let s be the number of switches in $\mathbf{N}$. In step 1 of $Main()$, we generate n packets with the $Seed(N)$ operation. In step 2, we generate another n packets with the $Switch?(N)$ operation. In step 3, we generate one packet. In step 4, for each switch, we

generate $(2 + s)$ packets. No SNMP traffic is generated by the $Topology()$ routine. Thus, the algorithm generates a total of $2n + 1 + s(s + 2)$ packets. If we consider that every switch has a fixed number of ports, then we may conclude that $n \propto s$. Thus, we generate $\mathcal{O}(s^2)$ packets.

Now let us use broadcast SNMP. Because the $Switch?(N)$ operation is independent of the agent queried, we can accomplish step 2 of $Main()$ with a single broadcast packet. Furthermore, because the $Fwd(S, M)$ operation is also independent of the agent queried, we can use broadcast SNMP in step 4 to generate only 3 packets for each switch. Thus, we now generate a total of $n + 2 + 3s$ packets. We have made the number of packets $\mathcal{O}(s)$ rather than $\mathcal{O}(s^2)$.

Note that we cannot use a broadcast packet to replace the $Seed(N)$ operation because we need R to generate ARP request packets for each $N \in \mathbf{N}$. This will only occur if we use unicast packets.

6 Synchronization

6.1 Motivation

The running time of the topology discovery algorithm can be substantially reduced by synchronizing the SNMP querying operations. The computation requirements of the algorithm are fairly low and the I/O requirements are relatively high; in the absence of synchronization, the majority of the running time will be spent waiting for SNMP packets to return.

This section describes two possible models for introducing synchronous SNMP operations to the $Main()$ routine of the topology discovery algorithm. The first does not use the broadcast SNMP technique described in the previous section; the second further optimizes the algorithm by combining broadcast SNMP with synchronization.

We assume an SNMP library that supports a non-blocking query operation. That is, the SNMP library should allow an application to create and send out an SNMP packet without blocking until a response is received. The replies arrive asynchronously and are passed back to the application by the SNMP library invoking callback routines. This model gives us some flexibility in application design. An individual thread, after transmitting an SNMP query, may either wait for a reply to its query (or a timeout) or terminate and allow the SNMP library to deliver the results later.

6.2 Synchronization without Broadcast SNMP

We can increase the performance of the $Main()$ through synchronization in several ways without using broadcast SNMP. We do not provide a formal description of these modifications, but their implementation is straightforward and should not affect the proof provided for the algorithm.

- We can begin by spawning a separate thread for each $N \in \mathbf{N}$ to generate and send the packets for the $Seed(N)$ and $Switch?(N)$ operations. The thread then exits.
- As we receive each reply to the $Switch?(N)$ operation, if the response is positive, we add N to $\mathbf{L}$ and spawn a new thread to generate the SNMP query for $ARP(R, N)$.
- We also create a thread to generate the SNMP query for $ARP(R, R)$.
- At this point, we wait for all extant SNMP queries to either time out or generate a response.
- Now we spawn a new thread for each switch in $\mathbf{L}$. This thread generates all of the $Fwd(R, x)$ SNMP packets and exits.
- We again wait for all extant SNMP queries to either time out or generate a response. At this point, we have collected all of our data and are ready to invoke $Topology(R, \mathbf{L})$.

6.3 Synchronization with Broadcast SNMP

Introducing the use of broadcast SNMP allows even greater synchronization because we no longer have to discover the complete list of switches before we begin querying their switch forwarding tables. Again, we do not provide a formal description of the modifications, but they are straightforward to implement and do not affect the algorithm in principle.

- We begin by spawning a separate thread for each $N \in \mathbf{N}$ to generate and send the packets for the $Seed(N)$ operation. The thread then exits.
- When all of those threads have terminated, we create a thread to generate a single broadcast SNMP query for the $Switch?(N)$ operation.

- As we receive each reply to the $Switch?(N)$ packet, if the response is positive, we add N to $\mathbf{L}$ and spawn a new thread to generate and send the SNMP query for $ARP(R, N)$.
- We also create a thread to generate the SNMP query for $ARP(R, R)$.
- As we receive each reply to the SNMP ARP query packets, we create a thread to generate and send a broadcast SNMP packet for the $Fwd(x, ARP(R, mac[N]))$ operation.
- We now wait for all extant SNMP queries to either time out or generate a response. At this point, we have collected all of our data and are ready to invoke $Topology(R, \mathbf{L})$.

7 Potential Obstacles

7.1 Explicitly Configured Uplink Ports

Historically, some Ethernet switches have been configured with explicit "network" or "uplink" ports. Such a port does not maintain switch forwarding information, as it is assumed to be connected to directly or indirectly to the router. If a packet with an unknown MAC address arrives on the network port, the switch will discard it rather than flood it out all ports and add it to the switch forwarding table.

This causes problems for the topology discovery algorithm, since it relies on being able to determine which switch port is on the operational switching path back to the router. We may be able to conclude that a switch has such an uplink port if $mac[R]$ is not in its switch forwarding table. Unfortunately, there does not exist a foolproof method for querying standard SNMP MIBs to discover which port this may be.

Because such ports are often located on the right side of the switch chassis, we may at least suggest the simple heuristic that in the absence of forwarding information for $mac[R]$, we guess that the uplink port is the highest-numbered port on the device. We can improve this heuristic at the expense of speed by guessing that the uplink port is the highest-numbered interface with an active link and no entries in the switch forwarding table.

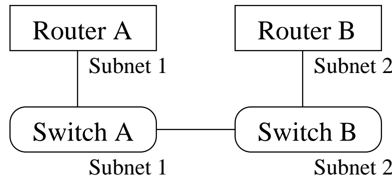
We feel obliged to note that "uplink" ports cause problems not only for the topology discovery algorithm, but also for the network as a whole. Their

behavior may save a substantial amount of switch forwarding memory, but can also result in the loss of network connectivity to hosts connected to the switch. For this reason, we strongly recommend that this feature not be used.

To see how this problem arises, consider the case in which a host A is connected to port x on a switch S with an explicitly configured uplink port y. When A transmits packets, S will associate A with port x in the switch forwarding table. Traffic from the network cloud arriving on port y will thus pass to port x. Now consider what happens if the entry for A in the switch forwarding table expires. Traffic from the network cloud arriving on port y will no longer pass directly to port x; neither will it be flooded out every port. Instead, the packet will be discarded.

7.2 Switches in Multiple Subnets

In [1], the authors raise concerns over some networks with multiple routers for which determining a unique network topology may not be possible. An example is shown in the figure below.



Router A is configured to be on subnet 1 and is directly connected to switch A, which has its management address on subnet 1. Router B is configured to be on subnet 2 and is directly connected to switch B, which has its management address on subnet 2. Switches A and B are connected. Router A is *not* configured for subnet 2, and router B is *not* configured for subnet 1.

Let us consider the problem this poses. If we begin our topology discovery from router A, we may have problems discovering that switch B even exists, since we may be unaware of subnet 2. We can attempt to resolve this situation by using the broadcast address of 255.255.255.255 for the $Switch?(N)$ operation. Switch B will respond to this request, but the response will return to H through router B.

This leaves the problem that switches A and B may not be in each other's switch forwarding tables. We need a way to force switch B to direct a packet

with its MAC address as a source toward switch A. At this time of this writing, we are still unsure how this may be accomplished.

On the other hand, note that this is an example of pathological network design. Because of the connection between the two switches, subnets 1 and 2 are necessarily part of the same broadcast domain, and both routers should be configured to reflect this.

7.3 Multiple Routers

Now consider the situation in which we have a set of two or more routers $\mathbf{R} = \{R_1, R_2, \ldots, R_k\} \subset \mathbf{N}$ and we need to determine the location of each router in the switching topology for $\mathbf{N}$.

This situation is not difficult to deal with. We first discover the topology tree from the perspective of R_i, where R_i is the router that passes traffic between H and hosts in $\mathbf{N}$. We then use the path discovery algorithm to locate every other router $R_j \in \mathbf{R}$ within the tree.

Note that if there are multiple connections between some R_j and the rest of $\mathbf{N}$, we will actually have a topology graph rather than a tree. In this case, at some step in the path discovery algorithm it will be true that $Fwd(T_i.root, mac[C])$ is defined and not equal to $uplink[T_i.root]$ for more than one subtree T_i. We simply modify the algorithm to take *both* paths in this case and tie the several paths together at the last hop.

7.4 Physical Topology Discovery

We also emphasize once more that the topology discovery algorithm determines the *operational* topology of the network, which consists solely of those switching links that are currently active. The physical topology of the network includes switching links that are inactive as a result of Spanning Tree Protocol; this links are invisible to the algorithm.

Similarly, the physical topology of the network includes any connections from switches to hosts that are not currently up and running. The path discovery algorithm is not capable of determining the switching path to these hosts once the switch forwarding table entries associated with it have expired.

7.5 Hubs and Unmanaged Switches

Hubs, repeaters, unmanaged switches, and switches for which we lack the proper management address or SNMP read access are all invisible to the topology discovery algorithm. They will appear in the topology tree as places where a wire appears to have more than two ends. For example, in the case where we find two switches connected to P_R, there is likely to be an "invisible" device between those switches and the router.

This property can actually be exploited if we want to locate places in the network where users may have installed their own network equipment. Because we know the switching topology for the network, we also know which active switch ports have connections to other switches and which have connections to hosts. If the switch forwarding table for an active port not involved in the switching topology consistently has multiple entries, we may assume that there is an "invisible" device connected to that port.

8 Implementation and Performance

Our reference implementation of the topology algorithm includes additional code for querying the *ifDescr* values of each switch port in the topology tree and does not include quite as much synchronization as the version described in the section above. It is written entirely in Java 1.3 and uses a lightweight SNMPv1 library that we developed to allow the use of broadcast SNMP and large-scale synchronization. Our implementation does not include any OS-specific hooks and has been tested under both Linux and Windows 2000.

Using Sun Microsystems' JRE with native thread support on a 1-GHz Pentium III system, we consistently are able to determine the switching topology of a network of 22 switches and 254 possible host addresses in about 2.5 seconds. With only 5 switches, the discovery time declines to about 1.4 seconds.

We expect that we may be able to reduce the running time further by pregenerating SNMP "template" packets and increasing the level of synchronization in our implementation. An additional speed improvement may come from implementing the algorithm in C with native thread libraries.

9 Conclusion

We have described a reliable topology discovery algorithm that focusses on the operational topology of the network. This algorithm is relatively inexpensive computationally; its running time is $\mathcal{O}(s^2)$, where s is the number of switches in the network. Its implementation is also straightforward and does not rely on any proprietary SNMP MIBs or require any privileged access to the network stack of the network management host.

We believe that this algorithm represents a refinement of previous research in this area in terms of both complexity and running time. We hope that it will encourage further work in creating diagnostic tools that examine not just the network layer of the protocol stack, but also the data link layer.

Our future work in automatic discovery of Ethernet topology is likely to focus on systems for detecting error conditions in the switching path to a host. If the topology algorithm is embedded in routers themselves—that is, if $H = R$—then it may be possible to develop a more powerful version of the standard *traceroute* utility that operates at the Ethernet layer rather than the IP layer.

We also seek to establish a standard XML Document Type Definition for the topology of a switched Ethernet network. We can then couple this DTD with an mechanism for anonymizing switch addresses and make topology information for a local network available to remote users in a secure fashion.

Finally, it may be possible to couple the algorithm with standard network layer topology discovery methods to produce a browsable map of an entire autonomous system in a short amount of time. If we anonymize this data and provide a mechanism for autonomous systems to connect the borders of their generated topologies, we can create the foundations for a dynamic map of the operational topology of the Internet as a whole.

10 Acknowledgements

Supported by the Data Network Services group of Indiana University and the Advanced Network Management Lab of the Pervasive Computing Laboritories at Indiana University.

References

[1] Y. Breitbart, M. Garofalakis, C. Martin, R. Rastogi, S. Seshadri, and A. Silberschatz. Topology discovery in heterogeneous IP networks. In *Proceedings of INFOCOM 2000*. ACM, 2000.

[2] B. Lowekamp, D. O'Hallaron, and T. Gross. Topology discovery for large ethernet networks. In *SIGCOMM '01*, pages 237–248. ACM, 2001.

[3] K. McCloghrie and M. Rose, editors. *RFC 1213: Network Management Base for Network Management of TCP/IP-based internets: MIB-II*. IETF Network Working Group, 1991.

[4] E. Decker, P. Langille, A. Rijsinghani, and K. McCloghrie. *RFC 1493: Definitions of Managed Objects for Bridges*. IETF Network Working Group, 1993.

[5] J. Case, M. Fedor, M. Schoffstall, and J. Davin. *RFC 1157: A Simple Network Management Protocol (SNMP)*. IETF Network Working Group, 1990.

[6] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. *RFC 1901: Introduction to Community-Based SNMPv2*. IETF Network Working Group, 1996.

[7] J. Case, R. Mundy, D. Partain, and B. Stewart. *RFC 2570: Introduction to Version 3 of the Internet-standard Network Management Framework*. IETF Network Working Group, 1999.

[8] CERT. *CERT Advisory CA-1998-01: Smurf IP Denial-of-Service Attacks*. Carnegie Mellon University, 1998.

[9] T. Bray, J. Paoli, C. Sperberg-McQueen, and E. Maler, editors. *Extensible Markup Language (XML) 1.0, Second Edition*. W3C XML Core Working Group, 2000.