Florida State University Libraries

Electronic Theses, Treatises and Dissertations | The Graduate School

2010

End-to-End Approachesfor Ethernet Switch Level Topology Discovery

Joshua Lawrence

Follow this and additional works at the FSU Digital Library. For more information, please contact lib-ir@fsu.edu

THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

END-TO-END APPROACHES FOR ETHERNET SWITCH LEVEL TOPOLOGY DISCOVERY

By

JOSHUA LAWRENCE

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Summer Semester, 2010

The members of the committee approve the thesis of Joshua Lawrence defended on July 6, 2010.

Xin Yuan
Professor Directing Thesis

Zhenhai Duan
Committee Member

Zhenghao Zhang
Committee Member

Approved:

Dr. David Whalley, Chair, Department of Computer Science

Dr. Joseph Travis, Dean, College of Arts and Sciences

The Graduate School has verified and approved the above-named committee members.

ACKNOWLEDGMENTS

I would like to extend a great debt of thanks to my major professor, Dr. Xin Yuan, for his support, patience, and advise while completing this thesis under his direction. I would also like to thank my family and friends for their continuous support and advise.

TABLE OF CONTENTS

List of Tables v

List of Figures vi

Abstract vii

1 Introduction **1**

2 Ethernet Switched LANs **4**

2.1 Ethernet Frame 4

2.2 Ethernet Switch 5

2.3 Ethernet Topology 7

2.4 Address Resolution Protocol 8

3 Automatic MPI Topology Discovery Tool **10**

3.1 Computing the Tree Topology from the hop-count matrix 11

3.2 Obtaining the Hop-Count Matrix 18

3.2.1 Raw RTT Measurements 18

3.2.2 Data clustering 19

3.2.3 Inferring the Hop-count Distance 20

3.3 Evaluation 23

4 Client-Side Ethernet Switch Location Tool **26**

4.1 Placement of LS daemons 27

4.2 Techniques and Location Algorithm 27

4.2.1 MAC Address Spoofing 28

4.2.2 End-to-End Ethernet Switch Training 28

4.2.3 Client-Side Testing 29

4.2.4 Location Algorithm 30

4.3 Evaluation 34

5 Related Work **37**

6 Conclusion **38**

References 39

Biographical Sketch 41

LIST OF TABLES

3.1 Total execution time for each configuration 25

3.2 The execution time for each phase (100 Mbps switches) 25

3.3 The execution time for each phase (1Gbps switches) 25

4.1 Number of messages for location 36

LIST OF FIGURES

2.1 Ethernet frame 5

2.2 Ethernet switch example 6

2.3 Ethernet MAC address spoofing 7

2.4 Spanning Tree Example 8

2.5 An example ARP request/reply 9

3.1 A General Topology Example 12

3.2 The connectivity of the related nodes in G_1 and G_2 14

3.3 An example for topology calculation 16

3.4 Algorithm for tree topology 17

3.5 A round-trip time matrix 19

3.6 Data clustering algorithm 21

3.7 A Hop-Count Matrix 22

3.8 Topologies for evaluation 23

4.1 Ethernet LAN with Location Services 27

4.2 Ethernet Switch Training 29

4.3 Main-LS responding with MAC_X ARP reply 30

4.4 Test Ethernet switches by tree height 32

4.5 Location Algorithm 33

4.6 Location algorithm example 34

4.7 Topologies for evaluation 35

ABSTRACT

We present two tools that utilize the end-to-end approach for topology discovery for Ethernet switched LANs. Unlike existing Ethernet topology discovery methods that rely on Simple Network Management Protocol (SNMP) queries to obtain topology information, our tools infer topology based on end-to-end message exchanges. This method allows our tools to work on Ethernet switched LANs that use managed and/or unmanaged Ethernet switches without requiring any special privileges. We discuss the key concepts, the algorithms used, and report the evaluation for each tool.

CHAPTER 1

INTRODUCTION

Ethernet is the most popular local area networking technology due to its low cost and high performance cost ratio. It provides the connectivity to many networks such as small-private LANs and low-end high performance computer (HPC) clusters. We use the term *Ethernet switched LAN* or *Ethernet switched cluster* to refer to a system with workstations/personal computers (typically considered as end-nodes) connected by Ethernet switches. Such systems are widely used in small businesses and university departments. An Ethernet switched cluster can be a dedicated HPC cluster or a “casual” cluster that is formed by random computers running the same application.

Ethernet switched LANs are typically maintained by network administrators. Many maintenance and trouble-shooting tasks are dependent on accurate knowledge of the switch level topology. The switch level topology information can also be used to improve the communication performance of applications in these clusters. Although the system administrators can know the switch level topology in theory, in practice, they often do not have such information due to the large amount of cabling in a typical machine room. This thesis addresses this issue by developing tools that can automatically derive the switch level topologies of Ethernet switched clusters. Since non-privileged users run applications on such systems and may require the switch level topology information, our tools utilize topology discovery methods that do not require special privileges.

Some topology discovery tools utilize IP-level topology discovery of routers to give network administrators the ability to manually discover Ethernet switches; however, they pro-

vide no method for topology discovery of Ethernet switches. Some hardware vendors provide proprietary topology discovery methods, but they require all components (e.g. Ethernet switches) to be from the same hardware vendor. This could be useful for clusters with switches from the same vendor, however, not all Ethernet switched LANs use switches from the same vendor.

The Simple Network Management Protocol (SNMP) allows SNMP-equipped Ethernet switches (managed switches) to be queried for specific network-information. Commercial products exist using SNMP: OpenView from HP and NetView from IBM/Tivoli; however, these products use proprietary data and often fail to discover some components at the Ethernet switch level [6]. Other tools/methods that utilize SNMP queries for topology discovery are described in [1, 3, 6, 8]. Although these SNMP methods prove to discover the Ethernet switch level topology, not all Ethernet LANs utilize managed switches; or if managed switches exist, most users are not privileged to obtain their information. Topology discovery for systems with unmanaged switches is a difficult problem since such switches are passive and have no direct communication with end-nodes. Note that unmanaged Ethernet switches are much cheaper than their managed counterparts and are deployed more widely than managed switches.

The *end-to-end* topology discovery approach where the topology is discovered based on end-to-end message exchanges (without the support from switches) is advocated in [2]. We develop two tools that discover the Ethernet switch level topology using the end-to-end approach. Both tools do not require Ethernet switches to support SNMP and thus work when managed and/or unmanaged switches exist.

- The first tool is a Message Passing Interface (MPI) program that automatically discovers the switch level topology of an Ethernet switched cluster based on end-to-end performance measurements. This tool does not require any special privilege: anyone who can run MPI programs on a set of machines (that form a cluster) can use our tool to discover the Ethernet switch level topology. The limitation of this tools is that it works best on homogeneous clusters with similar machines and switches, and may not be able to handle heterogeneous clusters.

- The second tool is a client-daemon program that locates the Ethernet switch that connects an end-node to an Ethernet switched LAN. This tool uses end-to-end communication and the forwarding behavior of Ethernet switches to infer the Ethernet switch level topology. We provide an interface that allows any end-node to determine the Ethernet switch that connects it to an Ethernet LAN. This tool works on an arbitrary Ethernet switched cluster. The techniques used are similar to those in [2]. However, unlike the system in [2], our tool does not require a daemon to run on each node in the system.

The following chapters are organized as follows: Chapter 2 discusses concepts for Ethernet switched LANs; Chapter 3 presents our MPI topology discovery tool; Chapter 4 presents our client-side topology discovery tool; Chapter 5 discusses the related work and Chapter 6 concludes the paper.

CHAPTER 2

ETHERNET SWITCHED LANS

An Ethernet switched LAN (*Ethernet LAN*) consists of workstations/personal computers (*computers*) connected by Ethernet switches. The connection between a computer and an Ethernet switch (or two Ethernet switches) in an Ethernet LAN is called a *link*. Links are full-duplex: computers connected to an Ethernet switch can send and receive at the full link speed simultaneously. When dealing with an Ethernet switched cluster (*Ethernet cluster*), it is considered to be homogeneous if and only if it has the same type of computers, links, and Ethernet switches; otherwise, it is considered to be heterogeneous. Before we explain the details of our topology discovery tools, we briefly describe important concepts of Ethernet LANs that allow our tools to determine the network-topology.

2.1 Ethernet Frame

The format of an Ethernet-frame is shown in Figure 2.1. The first two sets of 6-bytes are addresses that represent the destination and source, respectively, for an Ethernet-frame. The next 2-bytes specify whether an Ethernet-frame contains certain protocol information such as an IP-packet, an ARP-packet, or simply a raw Ethernet-frame. The next block of bytes represent the data carried in an Ethernet-frame. For example, if an Ethernet-frame is carrying an IP packet, the data section contains the IP-packet that is defined in the IP protocol. The data-block is at least 46 bytes but no more than 1500 bytes. For example, if an IP-packet is greater than 1500 bytes, then multiple Ethernet-frames are needed to send an IP-packet; on the other hand, if the size of an IP-packet is not at least 46 bytes, then

the data is padded with zeros. The final 4-bytes is the frame check sequence used to detect any errors to the Ethernet-frame during transfer.

An address in an Ethernet-frame is represented by a *MAC* address: a 48-bit identifier assigned to an Ethernet-adapter to uniquely identify an adapter in an Ethernet LAN; typically each computer has at least one Ethernet-adapter. By default, an Ethernet-adapter only accepts Ethernet-frames whose destination MAC address is the one assigned to that Ethernet-adapter; however, to force an Ethernet-adapter to accept foreign Ethernet-frames, it is placed in the *promiscuous* mode. We use the notation $\mathbf{MAC}_a$ to represent the MAC address of node a.

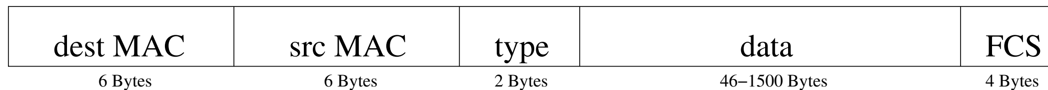

Figure 2.1: Ethernet frame

2.2 Ethernet Switch

Ethernet switches are transparent network-devices with multiple ports. A switch stores MAC addresses with their incoming-ports to a **forwarding table**. The forwarding table is used when a switch receives an Ethernet-frame to look up the destination-MAC and forwards the Ethernet-frame to the corresponding outgoing-port; if the destination-MAC does not exist in the forwarding table, the switch floods the Ethernet-frame to all outgoing-ports. Also, while looking for the destination-MAC, a switch stores the source-MAC and its corresponding incoming-port in the forwarding table. An Ethernet switch that uses the *store-and-forward* technique does not forward an Ethernet-frame until the entire frame arrives at the Ethernet switch; this process causes a distinguishable latency while an Ethernet-frame travels through an Ethernet switch. Our MPI tool uses this property to derive the switch level topology based on end-to-end performance measurements.

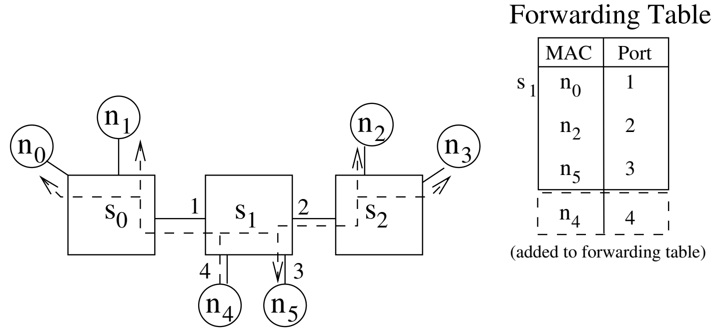

Figure 2.2: Ethernet switch example

Figure 2.2 shows computer n_4 sending an Ethernet-frame to computer n_3 through Ethernet switches. Currently, the forwarding table for s_1 contains mapping information for MAC$_{n_0}$ to port *1*, MAC$_{n_2}$ to port *2*, and MAC$_{n_5}$ to port *3*. We also assume s_0 and s_2 has no forwarding information for MAC$_{n_4}$ or MAC$_{n_3}$. When the Ethernet-frame arrives at s_1, it attempts to look up MAC$_{n_3}$ to see if it can forward the Ethernet-frame to a specific outgoing-port. In this case, the MAC address does not exist so the Ethernet-frame is forwarded to all outgoing-ports and the forwarding table updates MAC$_{n_4}$ by adding an entry that maps it to port *4*. When n_5 receives the Ethernet-frame, n_5 discards it. s_0 and s_2 run the same process as s_1 and flood the Ethernet-frame to all their respective outgoing-ports. Finally all nodes reject the Ethernet-frame except n_3. If n_3 decides to send an Ethernet-frame back to n_4, neither s_1 nor s_2 floods the Ethernet-frame since their respective forwarding tables have entries for MAC$_{n_4}$.

An Ethernet-frame is mainly built and modified by the software for an Ethernet-adapter; however, a superuser has the ability to build or modify the contents of an Ethernet-frame. For instance, a superuser can *spoof* the source-MAC address to trick other computers or Ethernet switches as to the origin of an Ethernet-frame. Thus, an Ethernet switch can forward an Ethernet frame to the wrong port if a computer is spoofing the source-MAC address. Figure 2.3 shows n_4 sending an Ethernet-frame to n_1, however, n_4 spoofs the source-MAC address with MAC$_X$. Consequently, this causes s_1 to update its forwarding table to map MAC$_X$ to port *4* instead of port *N*; now all Ethernet-frames destined for

MAC_X go to port *4*.

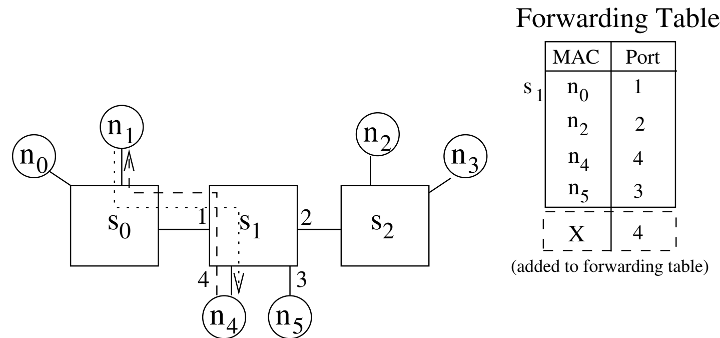

Figure 2.3: Ethernet MAC address spoofing

2.3 Ethernet Topology

Ethernet switches use a spanning tree algorithm to determine forwarding paths that follow a tree structure [15]; thus the switch level physical topology of the network is always a **tree**. Figure 2.4 shows an Ethernet LAN with a redundant link: s_0 has multiple paths to s_2. The spanning tree algorithm disables some links in the network such that the working links and switches form a tree topology. The dashed line in figure 2.4 represents a disabled link. This means the only way n_0 communicates with n_2 is through all three switches: $n_0 \rightarrow s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow n_2$

The topology can be modeled as a directed graph $G = (V, E)$ with nodes V corresponding to switches and computers, and edges E corresponding to unidirectional channels. Let S be the set of all switches in the Ethernet and M be the set of all computers ($V = S \cup M$). Let $u, v \in V$, a directed **edge** $(u, v) \in E$ if and only if there is a link between node u and node v. We call the physical connection between node u and v **link** (u, v), which corresponds to two directed edges (u, v) and (v, u). Since the network topology is a tree, the graph is also a tree with computers being leaves and switches being internal nodes. Notice that a switch may also be a leaf; however, such a switch does not participate in any

communication and thus is excluded from the topology. We define the *hop-count distance* between two end-computers to be the number of switches in the path between them (the path is unique in a tree).

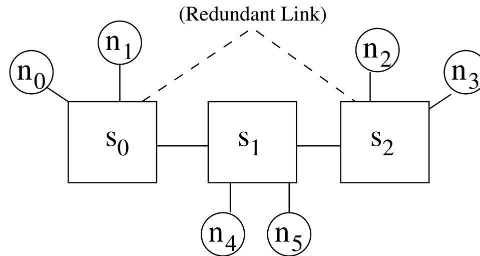

Figure 2.4: Spanning Tree Example

2.4 Address Resolution Protocol

Even though Ethernet uses a MAC address to send Ethernet-frames between computers, most computers are addressed by using an Internet Protocol (*IP*) address. To resolve an IP address to a MAC address, a computer uses the address resolution protocol (*ARP*). Once a computer resolves an IP address to a MAC address, it caches the mapping to an *ARP-cache* for faster retrieval. This caching prevents a computer from needing to run the ARP protocol every time it sends an Ethernet-frame; however, an ARP-cache entry does delete itself after a specified amount of inactivity.

As figure 2.5 shows, n_4 needs to send an Ethernet-frame to n_3. We assume n_4 has no ARP-cache entry for MAC_{n_3}. n_4 broadcasts an ARP-request asking for the owner of the destination-IP to respond with the corresponding destination-MAC address. The broadcast arrives at all computers in the Ethernet LAN; in this case, since n_3 is the beholder of the requested MAC, it sends and ARP-reply back to n_4 with MAC_{n_3}. Once n_4 receives the ARP-Reply, the corresponding IP and MAC are added to the ARP-cache; then an Ethernet-frame is built and wrapped around the IP-packet.

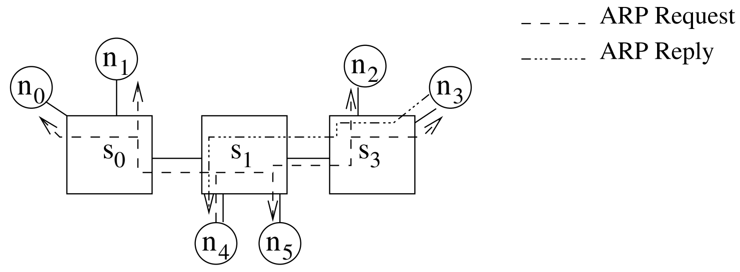

Figure 2.5: An example ARP request/reply

CHAPTER 3

AUTOMATIC MPI TOPOLOGY DISCOVERY TOOL

Many HPC applications running on Ethernet clusters are message passing programs that use Message Passing Interface (MPI) [9] routines to perform communications. MPI defined the API for message passing and is the industrial standard for developing portable and scalable message passing programs. Two types of communication routines are specified in MPI: point-to-point routines that perform communications between two processes and collective communication routines that carry out communications among a group of processors. For an Ethernet cluster to efficiently support MPI applications, it is essential that the MPI communication performance is maximized.

The physical switch level topology of an Ethernet cluster strongly affects the performance of MPI collective operations. In particular, in an Ethernet cluster with multiple switches, using topology specific communication algorithms can drastically improve the performance over the topology unaware algorithms used in common MPI libraries such as MPICH [10] and OPEN MPI [11]. To achieve high MPI performance, automatic routine generators that take the topology information as input and automatically produce topology specific MPI collective routines are developed [4, 5, 12, 13]. Unfortunately, for many clusters, especially the "casual" clusters, the switch level topologies are usually unknown. An automatic topology discovery tool can significantly benefit MPI applications on such systems.

We present and MPI topology discovery tool that automatically infers the switch level

topology from end-to-end measurements. Two key features in Ethernet clusters facilitate the determination of the switch level topology from end-to-end measurements.

- First, Ethernet switches use the store-and-forward switching mechanism. Thus, when a packet travels through a switch, a notable latency is introduced. Let the link bandwidth be B and the packet size be $msize$, the hardware store-and-forward latency is at least $2 \times \frac{msize}{B}$ ($\frac{msize}{B}$ for receiving the packet from the input port and $\frac{msize}{B}$ for sending the packet to the output port). Consider 1Gbps Ethernet switches. When sending a 1000-byte packet ($msize = 1000B$), each switch adds at least $\frac{2 \times 1000 \times 8}{10^9}s = 16\mu s$. Such a latency is reflected in the round-trip time (RTT) between each pair of machines. Hence, we can measure the RTT between the pairs of machines and derive the hop-count distance based from these measurements.

- Second, the switch level topology of an Ethernet cluster is always a tree [15]. When the hop-count distance between each pair of computers is decided, the tree topology can be uniquely determined (we will prove this and give an algorithm to compute the tree topology from the hop-count distances).

The topology discovery process in the tool consists of two steps. First, the end-to-end round-trip (RTT) between each pair of machines is measured and the hop-count distance is derived. The result of this step is a hop-count matrix with each entry in the matrix recording the hop-count distance between a pair of machines. Once the hop-count matrix is obtained, in the second step, the switch level (tree) topology is computed. In the following, we first discuss how to compute the network topology from the hop-count matrix, and then describe how to obtain the hop-count matrix.

3.1 Computing the Tree Topology from the hop-count matrix

Let the number of machines in the cluster be N and the machines be numbered from 0 to $N-1$. The hop-count matrix, HC, is an $N \times N$ matrix. Each entry in the hop-count matrix, $HC_{i,j}$, $0 \leq i \neq j \leq N-1$, records the hop-count distance from machine i to machine j.

Before we present the algorithm to compute the tree topology from the hop-count matrix, we show an example that a general topology (not a tree topology) cannot be uniquely

determined from the hop-count matrix. Consider the 3-machine cluster in Figure 3.1 (a). Let us assume that the paths are bi-directional and $path(0, 1) = 0 \rightarrow s_0 \rightarrow s_1 \rightarrow 1$; $path(0, 2) = 0 \rightarrow s_3 \rightarrow s_2 \rightarrow 2$; and $path(1, 2) = 1 \rightarrow s_1 \rightarrow s_2 \rightarrow 2$. Thus,

$$HC = \begin{bmatrix} 0 & 2 & 3 \\ 2 & 0 & 2 \\ 3 & 2 & 0 \end{bmatrix}.$$

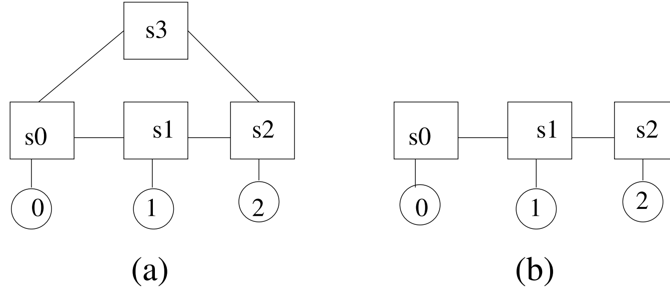

Figure 3.1: A General Topology Example

This matrix is also the hop-count matrix for the topology show in Figure 3.1 (b). This example shows that multiple (general) topologies can correspond to one hop-count matrix; hence, a hop-count matrix cannot uniquely determine a corresponding topology. Fortunately, this is not that case when the topology is a tree. Next, we show that the tree topology corresponding to a hop-count matrix is unique and can be derived from the hop-count matrix, which is the theoretical foundation of our tool.

Lemma 1: Let $G = (S \cup M, E)$ be a tree topology, S being the set of switches, M being the set of machines, and E being the set of edges. When $|S| \geq 2$, there exists a switch in $s \in S$ such that s is connected to exactly one other switch.

Proof: We will provide a constructive algorithm to find such a switch (internal node) that connects to only one other switch. Starting from any switch, s_0. If this switch connects only to one other switch, then $S = s_0$ is found; otherwise, pick any one of the switches connected to s_0. Let this switch be s_1. If s_1 connects only to one switch (s_0), then $s = s_1$

is found; otherwise s_1 connects to at least two switches. We will consider the switch that connects to s_1, but is not s_0. Let the switch be s_2. If s_2 connects to one switch (s_1), then the switch is found; otherwise, there are at least two switches connected to s_2 with one not equal to s_1. We can proceed and consider that switch. This process is then repeated. Since the topology is a tree, each time a new switch will be considered (otherwise, a loop is formed, which contradicts the assumption that the topology is a tree). Since there are a finite number of switches in the topology, the process will eventually stop with the switch that connects to only one other switch being found. □

Definition 1: Let M be the set of machines, tree topology $G_1 = (S_1 \cup M, E_1)$ is said to be equivalent to tree topology $G_2 = (S_2 \cup M, E_2)$ if and only if there exists a one-to-one mapping function $F : S_1 \cup M \rightarrow S_2 M$ such that (1) for all $u \in M$, $F(u) = u$; (2) for any $(u, v) \in E_1$, $(F(u), F(v)) \in E_2$; and (3) for any $(F(u), F(v)) \in E_2$, $(u, v) \in E_1$.

Theorem 1: Let M be the set of machines. Let $G_1 = (S_1 \cup M, E_1)$ and $G_2 = (S_2 \cup M, E_2)$ be two tree topologies connecting the machines M. Let $H(G)$ be the hop-count matrix for the topology G. If $H(G_1) = H(G_2)$, then G_1 is equivalent to G_2.

Proof: We prove this by induction on the size of S_1 (the number of switches in G_1). Base case, when there is only one switch in S_1. In this case, we have $H(G_1)_{i,j} = H(G_2)_{i,j} = 1$, $0 \leq i \neq j \leq N - 1$. If G_2 also has one switch, then G_1 and G_2 is equivalent since there is only one way to connect the N machines (each with one Ethernet card) to that switch. If G_2 has more than one switch, there are at least two switches in G_2 that are directly connected to machines (switches can only be internal nodes in the tree). Let s_1 and s_2 be two such switches in G_2 and let s_1 directly connect to machine m_1 and s_2 directly connect to machine m_2. We have $H(G_2)_{m_1,m_2} \geq 2$ since the path from machine m_1 to machine m_2 must pass through switches s_1 and s_2. This contradicts with the fact that $H(G_2)_{m_1,m_2} = 1$; hence, G_2 can only have one switch and is equivalent to G_1.

Induction case: assume that when $|S_1| \leq k$, when $H(G_1) = H(G_2)$, G_1 and G_2 are equivalent. Consider the cast when $|S_1| = k + 1$. From Lemma 1, we can find a switch

$s \in S_1$ that connects to only one other switch in G_1. This switch $s \in S_1$ must connect to at least one machine; otherwise, it is a leaf which is not allowed. Let the set of machines directly connect to s be M_s. Clearly, in G_2, there must exist a switch x directly connection to all machines in M_s; otherwise, $H(G_1) \neq H(G_2)$. Moreover, switch x must connect to one other switch in G_2. We will prove this statement by contradiction.

Assume that switch x connects to more than one other switch in G_2. Let switches y and z be two such switches that directly connect to x. Let machine $x_m \in M_s$ be a machine directly connected to switch x. Since switches are internal nodes, there must exist a machine whose path to x_m goes through switch y. Let this machine be y_m and the switch directly connected to y_m be switch ly. Similarly, there exists a machine whose path to x_m goes through switch z. We denote this machine as machine z_m and the switch it directly connects to be switch lz. Let us assume that in G_1, machine y_m directly connects to switch t and machine z_m directly connects to switch r. Machine $x_m \in M_s$ directly connects to switch s. in G_1. Figure 3.2 depicts the connectivity among these machines/switches in G_1 and G_2.

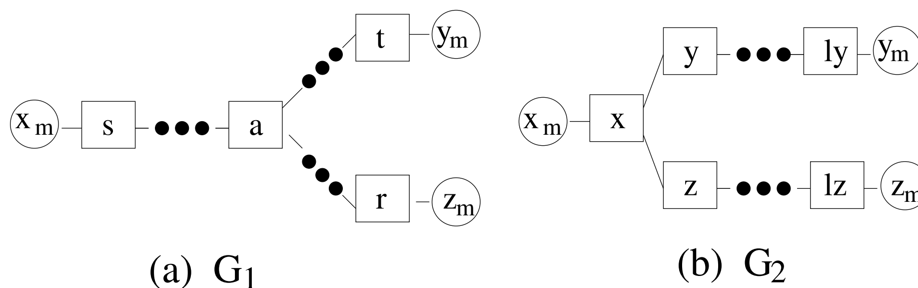

Figure 3.2: The connectivity of the related nodes in G_1 and G_2

In G_1 shown in Figure 3.2 (a), $path(x_m, y_m)$ and $path(x_m, z_m)$ share some switches (the first one is switch s). We denote the last switch in both paths be switch a. Hence, $path(x_m, y_m) = x_m \rightarrow s \rightarrow ... \rightarrow a \rightarrow ... \rightarrow t \rightarrow y_m$. $path(x_m, z_m) = x_m \rightarrow s \rightarrow a \rightarrow ... \rightarrow t \rightarrow z_m$. Since G_1 is a tree topology, there is a unique path between any two

nodes. Hence, the unique path between y_m and z_m is $path(y_m, z_m) = y_m \to t \to ... \to a \to ... \to r \to z_m$. Since s only connects to one other switch, $a \neq s$ and $H(G_1)_{y_m,z_m} < H(G_1)_{x_m,y_m} + H(G_1)_{x_m,z_m} - 1$.

In G_2 show in Figure 3.2 (b), $path(x_m, y_m) = x_m \to x \to y \to ... \to ly \to y_m$; $path(x_m, z_m) = x_m \to x \to z \to ... \to lz \to y_m$; and the unique path from y_m to z_m is $path(y_m, z_m) = y_m \to ly \to ... \to y \to x \to z \to ... \to lz \to z_m$. Hence, $H(G_2)_{y_m,z_m} = H(G_2)_{x_m,y_m} + H(G_2)_{x_m,z_m} - 1$. Under the assumption that $H(G_1) = H(G_2)$, we have $H(G_1)_{x_m,y_m} = H(G_2)_{x_m,y_m}$ and $H(G_1)_{x_m,z_m} = H(G_2)_{x_m,z_m}$. Hence,

$$\begin{aligned} H(G_1)_{y_m,z_m} &< H(G_1)_{x_m,y_m} + H(G_1)_{x_m,z_m} - 1 \\ &= H(G_2)_{x_m,y_m} + H(G_2)_{x_m,z_m} - 1 \\ &= H(G_2)_{y_m,z_m} \end{aligned}$$

This contradicts the fact that $H(G_1) = H(G_2)$. Hence, switch x can only connect to one other switch in G_2. Both G_1 and G_2 has a same sub-tree: $M_s \cup s$ in G_1 and $M_s \cup x$ in G_2. We can construct a reduced graph G_1^- by removing M_s from G_1 and changing switch s to a machine and a reduced graph G_2^- by removing M_s from G_2 and changing switch x to a machine. Clearly, $H(G_1^-) = H(G_2^-)$ when $H(G_1) = H(G_2)$. From the induction hypothesis, G_1^- and G_2^- are equivalent. Using the one-to-one mapping function f, that maps the switches in G_1^- to the switches in G_2^-, we can construct a one-to-one mapping function, g that maps switches in G_1 to the switches in G_2: $g(b) = f(b)$ for all $b \in S_1 - s$ and $g(s) = x$; and for all $u \in M$, $g(u) = u$. Clearly, (1) for any $(u, v) \in E_1$, $(g(u), g(v)) \in E_2$; and (2) for any $(g(u), g(v)) \in E_2$, $(u, v) \in E_1$. Thus, G_1 is equivalent to G_2; hence, when $H(G_1) = H(G_2)$, G_1 is equivalent to G_2. □

Theorem 1 states that the tree topology has a unique hop-count matrix. Next, we present an algorithm that computes the tree topology from the hop-count matrix. The idea of the algorithm is as follows: when the hop-count distances between all pairs of machines are 1s, the topology consists of one switch that connects to all nodes. When the hop-count distance for some pair of machines is larger than 1, there is more than one switch in the

topology. In this case, the algorithm finds one switch that connects to only one switch in the topology (Lemma 1), and constructs the sub-tree that consists of this switch and attached machines to this switch. After that, the algorithm reduces the size of the problem by treating the newly found sub-tree as a single machine (with one less switch than the original problem), re-computes the hop-count matrix for the reduced graph, and repeats the process until all switches are computed. This algorithm is shown in figure 3.4.

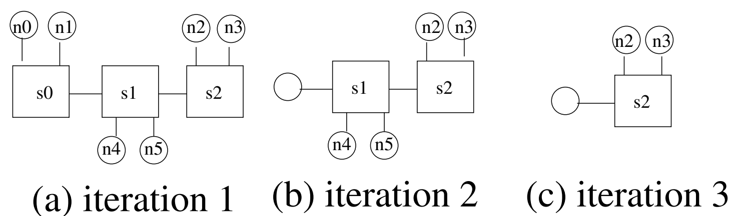

Figure 3.3: An example for topology calculation

Figure 3.3 how a topology is derived by the algorithm. The *WorkHC* stores the hop-count matrix in Figure 3.3 (a) in the first iteration. The nodes involved in the largest hop-count distance are connected to switch $s0$ or $s2$. Assume that a node connected to $s0$ is selected. The sub-tree consisting of n_0, n_1, s_0 is computed and the topology is reduced to Figure 3.3 (b). (*WorkHC* stores the hop-count distance for this topology in the second iteration). The newly added node is the blank circle. The hop-count distances for the reduced graph are derived from the old hop-count distances. For example, distance from the new node to n_4 is equal to the distance from n_0 to n_4 minus 1. In the next iteration, the sub-tree of n_0, n_1, s_0, n_4, n_5, and s_1 is discovered and is reduced to Figure 3.3 (c), which is discovered in the last iteration.

Input: the hop-count matrix, HC

Output: the tree topology stored in an

array $parent[..]$. Switches are numbered

from N to $N + |S| - 1$ in the result.

(1) $WorkHC = HC$; $size = N$;

(2) $for(i = 0; i < N; ++i)$ $NodeNum[i] = i$;

(3) $newswitch = N$;

(4) loop:

(5) $largestd = max_{i,j}\{WordHC_{i_j}\}$

(6) if ($largestd == 1$) then

/* the last switch left */

(7) for ($i = 0$; $i < size$; $++i$)

$parent[NodeNum[i]] = newswitch$;

(8) exit; /* done */

(9) end if

/* More than two switches left */

(10) Find a node n such that there exists a j,

$WorkHC_{n,j} = largestd$

(11) Find all nodes whose distances to n are 1s.

(12) Let the set of nodes be A.

(13) for all $i \in A$ parent[NodeNum[i]] = newswitch;

(14) Reduce the tree by removing all nodes in A

and adding a new leaf node newswitch,

recompute $WorkHC$, and $NodeNum$.

(15) ++newswitch;

(16) end loop

Figure 3.4: Algorithm for tree topology

3.2 Obtaining the Hop-Count Matrix

The algorithm in Figure 3.4 derives the topology from the hop-count matrix. The remaining issue is to obtain the hop-count matrix. In theory, it is fairly straight-forward to obtain the hop-count matrix from end-to-end measurements in a homogeneous cluster. The key issue is to filter out practical measurement anomalies; on the other hand, automatically deriving the hop-count matrix from end-to-end measurements for a heterogeneous cluster is a much more difficult problem that we are still testing. As a result, our tool works the best on homogeneous clusters. This tool uses three steps to obtain the hop-count matrix.

- Step 1: Raw RTT measurements. In this step, the tool uses *MPI_Send* and *MPI_Recv* to obtain the end-to-end measurements. The main challenge is to ensure that the empirical measurement results are sufficiently accurate.
- Step 2: Data clustering. Statistical methods are applied to filter out the noises in the RTT measurements and to group the raw RTT measurement results into classes with each class corresponding to a hop-count distance.
- Step 3: Hop-count inference. Hop-count distances are inferred from the classified RTT results.

3.2.1 Raw RTT Measurements

The RTT between each pair of machines is basically the ping-pong time between a pair of machines; hence, obtaining the RTT value for a pair of machines is straight-forward in theory. Since the accuracy of the empirical measurements has a very significant impact on the effectiveness of the tool, our tool incorporates several mechanisms to ensure the ping-pong time measurements are accurate

- RTT is measured with a reasonably large packet size. The default packet size in our tool is 1400 Bytes to ensure the latency from a 1Gbps (and 100 Mbps) is greater than the measurement noise.
- Each RTT measurement is obtained by measuring multiple iterations of message round trips and using the average of the multiple iterations. Measuring multiple iterations increases the total time and improves the measurement accuracy. The tools uses a parameter $NUMRTT$ to control the number of iterations in each RTT measurement.

- A statistical method is incorporated to ensure that the measured results are consistent. To determine the RTT between a pair of machines, a set of X RTT samples are measured. The average and the 95% confidence interval of the samples are computed. If the confidence interval is small (the ratio between the interval and the average is smaller than a predefined value), the results are consistent and accepted. If the confident interval is large, the results are inconsistent and rejected. In this case, the number (X) of samples to be measured is doubled when the measurement results are sufficiently consistent or when a predefined large number (1000) of samples are taken (in this case, the measurement results may still be inconsistent, but we use the results regardless). Having a threshold for the 95% confidence interval statistically guarantees the quality of the RTT measurements. The tool has two parameters related to the statistical method: $THRESHOLD$ that is used to determine whether the 95% confidence interval is sufficiently small, and $INITPINGS$ that is the number of samples taken initially.

The final measurement results are stored in the $RawRTT$ array at the sender. After all $RawRTT$ entries are measured, the results are gathered into machine 0, which performs the later steps. Figure 3.5 shows an example $RawRTT$ matrix that is obtained for the example topology in figure.

$$RawRTT = \begin{bmatrix} 0.0 & .131 & .157 & .156 & .173 & .175 \\ .134 & 0.0 & .159 & .158 & .176 & .177 \\ .160 & .155 & 0.0 & .132 & .158 & .159 \\ .159 & .159 & .132 & 0.0 & .159 & .159 \\ .176 & .176 & .159 & .159 & 0.0 & .132 \\ .176 & .175 & .159 & .156 & .132 & 0.0 \end{bmatrix}$$

Figure 3.5: A round-trip time matrix

3.2.2 Data clustering

As is seen from Figure 3.5, the raw RTTs collected in the first step have noises. For example, in Figure 3.5, *RawRTT[0][2] = 0.157ms* and *RawRTT[0][3] = 0.156ms* (as discussed earler, the difference in RTT values with different hop-count distances is at least

$16\mu s = 0.016ms$ assuming the packet size is 1000B). Hence, the two measurements should be treated as the same: the difference is due to measurement noises. The data clustering step tries to clean up the measurement noises and group the similar measurement results into the same class with each class covering a range of data values that correspond to a fixed hop-count distance.

The technique to locate these classes is traditionally known as *data clustering* [14]. In the tool, we use a simple data clustering algorithm show in figure 3.6. In this algorithm, the data are sorted first. After that, data clustering is done in two phases. The first phase (lines (4) to (15)), raw clustering, puts measurements that differ less than a threshold, ($measurementnoise$) into the same class. After raw clustering, in the second phase, the algorithm further merges adjacent classes whose inter-class distance is not sufficiently larger than the maximum intra-class distances. The inter-class distance is the difference between the centers of two classes. The maximum intra-class distance is the maximum difference between two data points in a class. In the tool, the inter class distance is at least $INTER_THRESHOLD$ times larger than the maximum intra-class distances (otherwise, the two adjacent classes are merged into one class). $INTER_THRESHOLD$ has a default value of 4. Note that standard data clustering algorithms such as the k-means algorithm [14] can also be used. We choose the simple algorithm because it is efficient and effective in handling data in our tool.

For the round-trip-time matrix in figure 3.5, the clustering algorithm groups the data into three classes: Class 0 covers range [0.131, 0.134]; Class 1 covers range [0.155, 0.160]; and Class 2 covers range [0.173, 0.177]. Finally, the RTT measurements in a class are replaced with the center of the class so they are numerically the same.

3.2.3 Inferring the Hop-count Distance

There are two ways that the hop-count distances can be inferred in the tool. When the hop-count distances between some machines in the cluster are known, the hop-count distances are correlated with the RTT times off-line. The RTT times (that correspond to hop-count distances) can be merged into the RawRTT-times during data clustering. In this

Input: RawRTT Matrix

Output: RTT Classes

```
(1) Sort all RTTs in an array, sorted[..]. The sorted
        array has N*(N-1) elements
(2) measurementsnoise = NUMRTT * 0.001
(3) numclass = 0;
/* initial clustering */
(4) class[numclass].lowerbound = sorted[0];
(5) prev = sorted[0];
(6) for (i=0; i < N * (N - 1); ++i)
(7)   if (sorted[i] - prev < measurementnoise)
(8)     prev = sorted[i]; /* expand the class */
(9)   else
(10)    class[numclass].upperbound = prev;
(11)    ++numclass;
(12)    class[numclass].lowerbound = sorted[i];
(13)    prev = sorted[i];
(14)  end if
(15) end for
(16) do
(17)  for each class[i]
(18)    if ((intra-distance(i) is less than
        inter-distance(i, i+1))/INTER_THRESHOLD)
(19)      Merge class[i] and class[i+1]
(20)    end if
(21)  end for
(22) until no more merge can happen
```

Figure 3.6: Data clustering algorithm

case, the hop-count distance for a class can be assigned to the known hop-count distance that corresponds to the RTT-time inside the class. The hop-count distances are determined when data-clustering is finished. This approach can be applied when the hop-count distances of some machines in the cluster are known, which is common in practice.

When the tool does not have any additional information, it uses a heuristic to infer the hop-count distance automatically. First, the heuristic computes the gaps between the center of adjacent classes. For the example from the previous section the gap between Class 1 and Class 0 is $0.157 - 0.132 = 0.025$; and the gap between Class 2 and Class 1 is $0.175 - 0.157 = 0.018$. The tool then assumes that the smallest gap, 0.018 in this example, is the time introduced by one switch. The tool further assumes that the smallest time measured corresponds to the hop-count of one switch (two machines connecting to one switch): $class[0].hops = 1$. The formula to compute the hop-count for other classes is

$$class[i].hops = class[i-1].hops + \lfloor \frac{class[i].center - class[i-1].center + \frac{smallest_gap}{2}}{smallest_gap} \rfloor$$

In the example, we have $smallest_gap = 0.018$. $class[1].hop = 1 + \lfloor \frac{0.157-0.132+0.018/2}{0.018} \rfloor = 2$, and $class[2].hops = 2 + \lfloor \frac{0.175-0.157+0.018/2}{0.018} \rfloor = 3$. Hence, the hop-count matrix for this is shown in Figure 3.7.

$$HC = \begin{bmatrix} 0 & 1 & 2 & 2 & 3 & 3 \\ 1 & 0 & 2 & 2 & 3 & 3 \\ 2 & 2 & 0 & 1 & 2 & 2 \\ 2 & 2 & 1 & 0 & 2 & 2 \\ 3 & 3 & 2 & 2 & 0 & 1 \\ 3 & 3 & 2 & 2 & 1 & 0 \end{bmatrix}$$

Figure 3.7: A Hop-Count Matrix

The heuristic does not work in all cases. It works when (1) there exists one switch connecting more than one machine; and (2) there exist machines that are two switches apart. Most practical systems consist of a small number of switches and satisfy these two conditions.

3.3 Evaluation

We evaluate the topology discovery tool on homogeneous clusters with various configurations. Figure 3.8 shows the topologies used in the evaluation. We refer to the topologies in Figure 3.8 as topologies (1), (2), (3), (4), and (5). Topology (1) contains 16 machines connected by a single switch. Topologies (2), (3), (4), and (5) are 32-machine clusters with different network connectivity. The machines are Dell Dimension 2400 with a 2.8 GHz P4 processor, 640MB of memory, and 40GB of disk space. All machines run Linux (Fedora) with 2.6.5-1.358 kernel. The Ethernet card in each machine is a Broadcom BCM 5705 1Gbps/100Mbps/10Mbps with a Broadcom driver. Both giga-bit Ethernet (1Gbps) and fast Ethernet (100Mbps) switches are used in the test (each configuration contains one type of switch). The giga-bit switches are Dell Powerconnect 2624 (24-port 1Gbps switches) and the fast Ethernet switches are Dell Powerconnect 2224 (24-port switches).

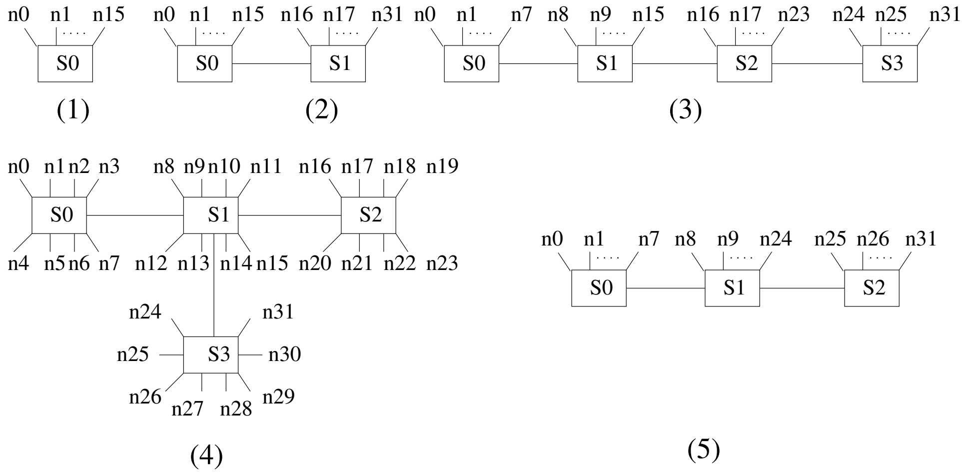

Figure 3.8: Topologies for evaluation

In the evaluation, we have defined the following parameters. The RTT measurement scheme has a packet size of 1400 Bytes; *NUMRTT=5*; *INITPINGS=26*; the 95% confidence interval threshold is 3% of the average. In the data clustering scheme, the noise thresh-

old is *measurementnoise=0.001*NUMRTT* (NUMRTT micro-seconds) and the inter-cluster distance is *INTER_THRESHOLD=4* times the maximum intra-cluster distance.

The tool successfully recognizes all the topologies in Figure 3.8 with both 1Gbps and 100Mbps switches, which demonstrates that the tool is robust in discovering switch level topologies. In the following, we present timing results for the tool. Each result is an average of five executions. All times are in the unit of second.

Table 3.1 shows the total execution time of the tool for each configuration in Figure 3.8. The second column contains the results with 100Mbps switches; and the third column are the results with 1Gbps switches. As can be seen from the table, it takes about 120 seconds to discover the topology with 32 nodes connected by 100Mbps switches and less than 35 seconds for 1Gbps switches. We further breakdown the execution time of each phase in the tool in Tables 3.2 and 3.3, which show the times for raw RTT measurement (RTT), data clustering and hop-count inferring (DCHC), and topology calculation (TC). As seen from the table, for all cases, more than 99.9% of the total is spent on the RTT measurements phase. This is due to the $O(N^2)$ entries in the RTT matrix that need to be measured and the extensive statistical techniques that are incorporated in the tool to ensure the accuracy of the measurement results. If there is a need to improve the execution time, one can focus on optimizing the RTT measurement methods in this tool. In the experiments, most ($> 99\%$) of the RTT measurements take 26 samples to reach the confidence interval threshold. Few measurements require 52 samples. There is no measurement that requires more than 104 samples. The DCHC and TC phases run on machine 0 and do not involve communications. When N is small, the time for these two phases is insignificant.

Table 3.1: Total execution time for each configuration

Topology	100Mbps	1Gbps
(1)	20.200s	6.935s
(2)	101.812s	31.368s
(3)	126.741s	34.940s
(4)	122.377s	33.436s
(5)	110.007s	32.248s

Table 3.2: The execution time for each phase (100 Mbps switches)

Topology	RTT	DCHC	TC
(1)	20.188s	.007s	.005s
(2)	101.792s	.013s	.008s
(3)	126.716s	.056s	.010s
(4)	122.351s	.016s	.010s
(5)	109.987s	.013s	.008s

Table 3.3: The execution time for each phase (1Gbps switches)

Topology	RTT	DCHC	TC
(1)	6.926s	.005s	.004s
(2)	31.350s	.010s	.007s
(3)	33.900s	.022s	.020s
(4)	33.406s	.020s	.011s
(5)	32.231s	.010s	.007s

CHAPTER 4

CLIENT-SIDE ETHERNET SWITCH LOCATION TOOL

Many topology discovery tools for Ethernet LANs set out to discover all network-components by involving all end-nodes in the discovery process. However, although most Ethernet LANs do not frequently alter the Ethernet switch structure, they may constantly have changes in end-nodes: machines in Ethernet switched LANs may be turned on and off. This *location tool* assumes that the switch level topology is fixed and known, and discovers the switch level topology for an arbitrary set of machines (a cluster) connected by Ethernet switches by having each machine in the cluster identify its location (switch) in the network.

The location tool consists of two programs: the **location service** (LS) daemon, and a client-side program. One LS is designated as the main-LS to act as *central* control for all LSes since an Ethernet LAN can contain multiple LSes. This tool assumes knowledge of the Ethernet switch level topology of the whole system, which can be obtained by the techniques in [2]. Each LS daemon, which maintains the switch level topology of the whole system, is placed in a vantage point in the network to provide the location service for arbitrary end-nodes, that is, to locate the Ethernet switch that is connected to any given machine. In comparison to our MPI tool described in chapter 3, the location tool works on all Ethernet LANs and does not require the use of MPI; however, it does require the use of our daemon in some nodes. The client-side program is a regular user program that a non-privilege user can run to discover its location.

4.1 Placement of LS daemons

All LS daemons agree on a tree topology of the network. The placement of LSes guarantees that an end-node connected to any switch can be detected and is done as follows. First, each leaf switch has a LS running on a node directly connected to the switch. For each internal switch, there are at least two LSes running on two branches of its descendants. All LS daemons know the switch topology as well as the locations of all other LS daemons. Figure 4.1 (a) shows an example Ethernet LAN with an LS connected to s_0 and s_2; notice that an LS is not connected to s_2. Figure 4.1 (b) shows the same cluster as a tree (this is how main-LS views the topology) and also points out the difference between an internal-switch (s_1) and a leaf-switch (s_0 and s_2). The client-side program is executed from any computer (such as n_4 in Figure 4.1(a)) and does not require any special privileges to participate in the location algorithm. This means the client-side program can be used on any Ethernet LAN that utilizes our location service.

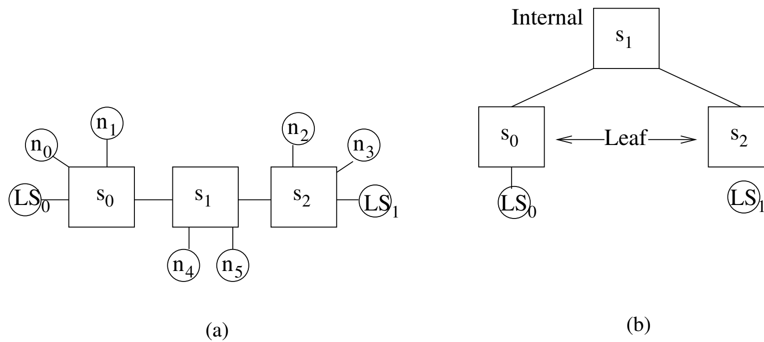

Figure 4.1: Ethernet LAN with Location Services

4.2 Techniques and Location Algorithm

Our location algorithm uses communication between the client-side program and LSes to allow a machine to identify its location on the network by (1) manipulating Ethernet switch forwarding tables with an arbitrary MAC address, and (2) testing the forwarding

paths using an IP packet destined for the arbitrary MAC address. Our location algorithm uses similar techniques as described in [2]. For the location algorithm to work correctly, we assume the Ethernet switch level topology along with the LS placement is provided to the main-LS. The main-LS stores the topology as a tree-structure for our location algorithm. Each LS is uniquely identified by an LS_{ID}. All LSes run in the promiscuous mode since our location algorithm utilizes arbitrary MAC addresses. We use the term Ethernet switch *training* to mean we force an Ethernet switch to add a MAC address and corresponding incoming port to its forwarding table. In the following, we first explain the techniques our location algorithm utilizes and then describe the location algorithm.

4.2.1 MAC Address Spoofing

Deriving the Ethernet topology from an arbitrary machine (for our tool) requires manipulating and testing forwarding paths in an Ethernet LAN. During the location algorithm, an LS spoofs source-MAC addresses to train forwarding paths since it can not use an existing MAC address without disrupting normal network-communication; thus, we use a pool of private MAC addresses that alternate through each run of the location tool to allow older addresses to expire from Ethernet switches. We call this arbitrary MAC address: $\mathbf{MAC}_X$. We represent communication (either Ethernet frame or IP packet) by using the notation $\mathbf{A : B \rightarrow C}$ where A is the source, B is the source address (which can potentially be spoofed), and C is the destination address. We need to specify the source address of A since an LS uses the ability to spoof B.

4.2.2 End-to-End Ethernet Switch Training

End-to-end Ethernet switch training trains all Ethernet switches between LS_i and LS_j with source address MAC_X. Once all Ethernet switches between LS_i and LS_j are trained, they forward MAC_X towards LS_i. In our tool, LS_i and LS_j can be the same LS (local training), and each LS has a special arbitrary local MAC-address. The location algorithm uses two end-to-end Ethernet switch training techniques.

- First, *local* training occurs when an LS trains a leaf Ethernet switch. An LS can not send a message to itself by using its own MAC address because the Ethernet frame

is internally looped-back; so each LS is assigned a unique local MAC address called $\mathbf{MAC}_{LS_a}$. Figure 4.2 (a) shows the process of an LS locally training Ethernet switch s_i. First, LS_m : $\mathrm{MAC}_{LS_m} \rightarrow$ BCAST is sent to ensure LS_m's Ethernet switch learns its local address. Even though this address leaks out to all other Ethernet switches, once s_i receives an Ethernet frame with destination MAC_{LS_m}, it forwards the frame back to LS_m. Also, since each LS is assigned its own MAC_{LS_m}, no other device should send an Ethernet frame with this address as its source. Once MAC_{LS_m} is trained, s_i can now be trained with MAC_X by sending LS_m : $\mathrm{MAC}_X \rightarrow \mathrm{MAC}_{LS_m}$. Now, when s_i receives an Ethernet frame destined for MAC_X, it is forwarded to LS_m.

- Second, *LS-to-LS* training occurs when an LS trains an internal Ethernet switch. Figure 4.2 (b) shows the process of an LS-to-LS training for Ethernet switches between LS_m and LS_n. LS_m : $\mathrm{MAC}_X \rightarrow \mathrm{LS}_n$ is sent and trains each Ethernet switch between s_i and s_k with MAC_X. If any Ethernet switch in path(s_i, s_k) receives an Ethernet frame with destination MAC_X, it is forwarded towards LS_m.

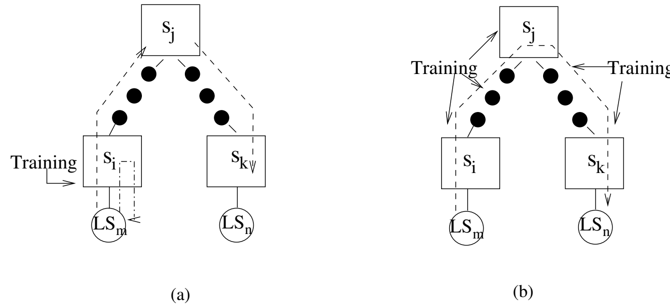

Figure 4.2: Ethernet Switch Training

4.2.3 Client-Side Testing

The client-side program is responsible for testing the forwarding behavior of MAC_X by sending a test IP packet to destination MAC_X. Since the client-side program works on the IP-level, it must receive its destination MAC address (MAC_X) by using ARP; however, since MAC_X is arbitrary, the client-side program is going to wait indefinitely for an ARP reply. To ensure the client-side program has the ability to send to MAC_X, we map an arbitrary IP address to it called IP_X. The main-LS is responsible for keeping the client-side program's

ARP-cache updated with the mapping of MAC_X to IP_X by sending it ARP replies during the location process. Figure 4.3 shows the process of n_4 trying to send a message to IP_X. First, n_4 sends out an ARP-request looking for the MAC address of IP_X; but, this is an arbitrary MAC address so no ARP response is sent; however, the ARP request is artificially answered by the main-LS (LS_0) by sending it MAC_X. Once n_4's ARP-cache is updated with the correct mapping, it now has the ability to send an IP test packet: $n_4 : \mathrm{IP}_{n_4} \rightarrow \mathrm{IP}_X$.

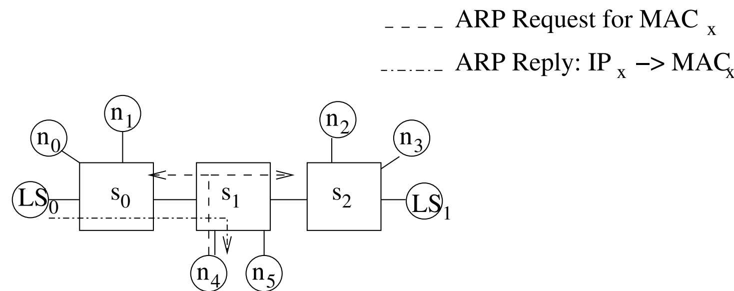

Figure 4.3: Main-LS responding with MAC_X ARP reply

Once the IP test packet is sent, the client-side program waits to receive replies from the LSes. The client-side program has two ways of interpreting these replies: (1) a reply from one LS indicates the client-side test has located its connected switch, and (2) a reply from two or more LSes indicates the Ethernet switch has not been located. At this point, though, the client-side program does not know its exact Ethernet switch location.

4.2.4 Location Algorithm

Now that we have the ability to train and test forwarding paths, we will describe the location algorithm. The location algorithm describes how the client-side program initiates contact with the main-LS and how the LSes facilitate the location discovery. The location algorithm consists of two main steps.

- Step 1: Initialization. Client-side program attempts to contact main-LS.

- Step 2: Path Forwarding Test. Each height-level in the tree is trained and tested to locate the client's connected Ethernet switch.

Initialization. The client-side program initiates the location algorithm by broadcasting a "hello" message. The main-LS is responsible for listening for this message and responds back to the client-side program with an acceptance message and IP_X (the IP address used for testing). The client-side program now waits until the main-LS issues him more commands as described in the location algorithm. Once the initialization is complete, the main-LS begins the main part of the location algorithm.

Path forwarding test. The path forwarding test utilizes end-to-end Ethernet switch training combined with the client-side test to locate a client's Ethernet switch. The switches are tested in a bottom-up fashion. All switches in the same height are trained at the same time by using the end-to-end training method on each switch. The switches are then tested with the client-side test to determine whether the client is connected to one of the switches. The order in which the Ethernet switches are tested correlates to their height-levels in the tree. Figure 4.4 shows how the main-LS sets the tree height-levels and the order that each height is tested (indicated by the arrow). The Ethernet switches need to be tested starting from height two to the maximum tree height. This prevents Ethernet switches above the current tested height from knowing where to forward MAC_X; since the LSes that are used to train the Ethernet switches are always found on the branches of an Ethernet switch and not from the parent.

The location algorithm, which involves all LSes and the client, is shown in figure 4.5. In this algorithm, the topology is structured as a tree, T. In the the first iteration, *crntHeight* is set to 2 and the algorithm pulls all the switches at *crntHeight* from T so they can be trained. Then, a sending LS and a receiving LS are used to train the path (LS_{send}, LS_{recv}) for each switch at *crntHeight* (lines (6) - (16)). Once training is finished, the main-LS tells the client-side program to send the test IP packet and gather replies from the responding

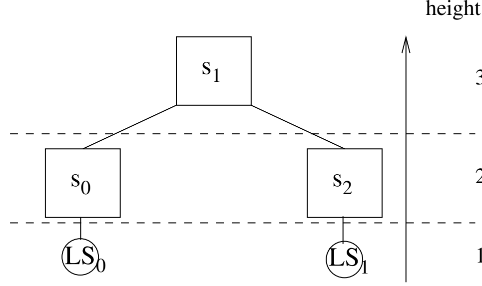

Figure 4.4: Test Ethernet switches by tree height

LSes (lines (17)-(18)). If there is only one reply, the client finds its location. If there are multiple LS responses then the client does not locate its switch and the algorithm continues at $crntHeight{+}1$. This process continues until either the switch is located or all tree-heights are exhausted. In which case, the client is connected to the root.

Let the number of switches in the whole system be S, the number of leaf-switches be L, the number of LSes be LS, and the tree height be H. The total number of control messages to locate a client is at most $2S + 2L + (S - L) + H + H \times LS$: for each switch, the main-LS needs to send and receive a message to the sender of training ($2S$); for each leaf-switch, two training messages must be sent ($2L$); for each internal switch, one training message must be sent ($S - L$); for each tree height, the client sends one message (H); for each tree height, the client at most receives LS replies from LSes ($H \times LS$).

Figure 4.6(a) shows an example run of the location algorithm. This example starts after the client has already initialized the location algorithm by contacting the main-LS and only shows the messages that are used to train and test the Ethernet switches. In figure 4.6(a), the main-LS starts the location algorithm by pulling all switches at height 2 (s_0). The main-LS looks for a sending and a receiving LS for s_0; however, in this case, this switch is a leaf-switch so it is locally trained by its respective LS (LS_0). Once the training is finished, the main-LS tells n_0 to send a test packet to IP_X. Since s_3 does not know where to forward MAC_X, it floods the test packet. This means LS_0, LS_1, and LS_3 receive the test IP packet

Input: Ethernet topology tree, T

(1) /* start at height 2 of T */

(2) $crntHeight = 2$

(3) $found = false$

(4) do

(5) $switches[..]$ = get Ethernet switches at $crntHeight$ in T

(6)   for each $switches[i]$

(7) LS_{send} = DFS_LEFT($switches[i]$)

(8) LS_{recv} = DFS_RIGHT($switches[i]$)

(9)   main-LS sends message to LS_{send} to start training

(10)   if LS_{recv} = NULL then // leaf switch

(11) LS_{send} sends local training messages

(12)   else // internal switch

(13) LS_{send} sends LS-to-LS training message to LS_{recv}

(14)   end if

(15) LS_{send} sends a message to main-LS to indicate the completion of training

(16)   end for

(17)   main-LS tells client-side program to send test IP packet

(18) $recieved$ = client-side program's results

(19)   if $received.numReplies = 1$ then

(20)   client located at $received.\mathrm{LS}_{ID} \in switches[i].\mathrm{LS}_{send}$

(21)   return switches[i]

(22)   end if

(23)   ++crntHeight

(24) while $crntHeight < T.height$ or $!found$

(25) return root

Figure 4.5: Location Algorithm

and respond back to n_0. Since this is more than one reply, n_0 sends a not found message back to the main-LS.

Figure 4.6(b) starts training at height 3 (s_1 and s_2). The main-LS searches for a sending and receiving LS for s_1 since it is an internal-switch; LS_0 and LS_1 are found and then used to train path(LS_0, LS_1); s_2 is leaf-switch and is thus locally trained with LS_2. The client sends the test IP packet and s_3 still forwards the packet to s_1 and s_2; however, s_1 forwards the IP packet to s_0. Both LS_0 and LS_2 receive the test packet and respond back to n_0. Since this is a multiple message response, n_0 sends a not found message to main-LS. Next, the training starts at height 3, however, this is the root of the tree meaing n_0 is connected to s_3.

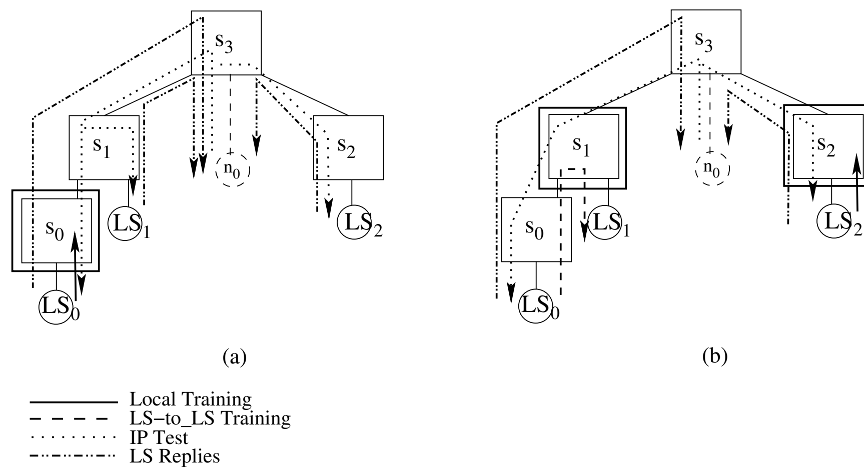

Figure 4.6: Location algorithm example

4.3 Evaluation

The location tool was evaluated on Ethernet LANs with various configurations. Figure 4.7 shows the Ethernet switch topologies with the attached LSes. We refer to the topologies as (1), (2), and (3). Topology (1) contains a single switch with a single LS. Topology (2) contains two switches with an LS connected to both switches. Topology (3)

contains three switches with an LS connected to s_0 and s_2. Each LS runs on a Dell Dimension 2400 with a 2.8 GHz P4 processor, 640MB of memory, and 40GB of disk space. Each machine runs Linux (Fedora) with 2.6.5-1.358 kernel. The Ethernet card is Broadcom BCM 5705 1Gbps/100Mbps/10Mbps card with the driver from Broadcom. Both Giga-bit Ethernet (1Gbps) and fast Ethernet (100Mbps) switches are used in the test. The Giga-bit switches are Dell Powerconnect 2624 (24-port 1Gbps Ethernet switches) and the fast Ethernet switches are Dell Powerconnect 2224 (24-port 100Mbps Ethernet switches). The client-side program was ran on a Dell Latitude D620 with a 2.00GHz Intel Core 2 processor, 1GB of memory, and 15GB of disk space. The operating system is Ubuntu 9.04 with 2.6.28-18-generic kernel. The Ethernet card is a Broadcom NetXtreme with the driver from Broadcom.

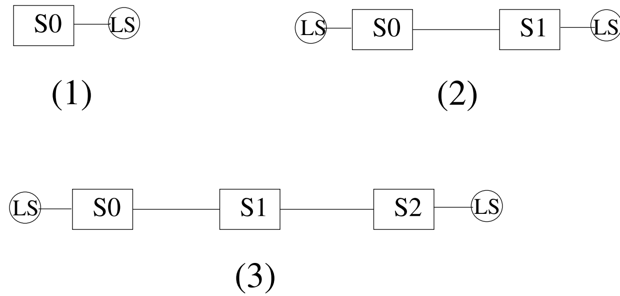

Figure 4.7: Topologies for evaluation

In the evaluation, we first discovered the Ethernet switch level topology for each topology in Figure 4.7 by using our MPI topology discovery tool. This information was used to create the Ethernet switch level topology information with the LS placements for the main-LS to use in the location algorithm. Each topology in Figure 4.7 was tested with each configuration containing the same type of switch and then tested with a mixture of switch types. (1Gbps and 100Mbps Ethernet switches used for the same configuration). We evaluated the location tool by connecting the client program to each Ethernet switch for each topology in Figure 4.7.

The tool successfully recognizes all the topologies in Figure 4.7. The client was able to locate its Ethernet switch and derives the topology in less than 1 second for all configurations; however, the number of messages it takes to locate the client's Ethernet switch varies by the height-level of an Ethernet switch. Since the location algorithm tests Ethernet switches from bottom-up, it takes more messages to discover the location of a client at higher levels in the tree-topology. Table 4.1 shows a breakdown of the number of messages it takes for a client to locate its Ethernet switch for the topologies in Figure 4.7. The second column represents the number of messages needed when the client is connected to s_0; the third column represents the same information for s_1; the fourth column represents the same information for s_2. The total number of messages is the accumulation of all end-to-end Ethernet switch training messages, the client test packets, and the LS replies. As the table shows, when a client is connected to s_0 in topologies (1) and (2), it only needs 9 messages to locate his Ethernet switch since s_0 is the only Ethernet switch at height 2; however, topology (3) has two Ethernet switches at 2, so the total number of messages includes the end-to-end training messages for both switches (s_0 and s_1), plus the client test packet, plus the LS reply from s_0. The table also shows how internal switches (such as s_1 in topology (2) and s_2 in topology (3) require more messages to locate the client's switch due to a higher height-level in the tree.

Table 4.1: Number of messages for location

Topology	s_0	s_1	s_2
(1)	9	N/A	N/A
(2)	9	17	N/A
(3)	11	11	20

CHAPTER 5

RELATED WORK

Determining the switch level topology in an Ethernet cluster is a challenging and important problem. Most existing methods [1, 3, 6, 8] and commercial tools [7, 16] rely on the Simple Network Management Protocol (SNMP) to query the Management Information Base (MIB) such as the address forwarding tables in the switches in order to obtain the topology information. These techniques are mainly designed for system management and assume the system administrator privilege. Moreover, they also require the switches to support SNMP, which is a layer-3 protocol and is not supported by simple layer-2 switches.

In [2], the authors discusses the problems with the SNMP based approaches in general and advocated an end-system based approach. The technique proposed in [2], however, is also designed with the assumption that the system administrative privilege is granted. Moreover, the techniques to derive the topology in [2] are totally different from ours.

Due to various difficulties in topology discovery, none of the existing methods can reliably discover topologies in all cases. In this paper, we present new methods that (1) allow a regular user to derive the system topology for homogeneous clusters where MPI applications run, and (2) a protocol that allows a client to discover its Ethernet switch location in any type of Ethernet switched cluster.

CHAPTER 6

CONCLUSION

In this paper, we introduce an MPI topology discovery tool for Ethernet switched clusters. The tool automatically infers the topology from end-to-end measurements. We also introduce a tool for determining a client location in an Ethernet. We describe the theoretical foundation for the MPI tool and also describe the algorithms for both tools in detail. Both tools are proved to be effective in discovering the topology of a network. Also, the MPI tool proved to be effective in producing efficient topology specific MPI collective routines.

REFERENCES

[1] Y. Bejerano, Y. Breitbart, M. GaroFalakis, R. Rastogi, "Physical Topology Discovery for Large Multi-subnet Networks", *IEEE INFOCOM*, pp.342-352, April 2003.

[2] R. Black, A. Donelly, C. Fournet, "Ethernet Topology Discovery without Network Assistance." *IEEE ICNP*, 2004.

[3] Yuri Breitbart, Minos Garofalakis, Ben Jai, Cliff Martin, Rajeev Rastogi, and Avi Silberschatz, "Topology Discovery in Heterogenous IP Networks: The *NetInventory* System", *IEEE/ACM Transactions on Networking*, 12(3):401-414, June 2004.

[4] A. Faraj, X. Yuan, P. Patarasuk, "A Message Scheduling Scheme for All-to-all Personalized Communication on Ethernet Switched Clusters," *IEEE Transactions on Parallel and Distributed Systems*, 18(2):264-276, 2007.

[5] A. Faraj, P. Patarasuk, and X. Yuan, "Bandwidth Efficient All-to-all Broadcast on Switched Clusters," *International Journal of Parallel Programming*, Accepted.

[6] Hasan Gobjuka, Yuri Breitbart, "Ethernet Topology Discovery for Networks with Incomplete Information," *IEEE ICCCN*, pp.631-638, Aug. 2007

[7] HP OpenView. Web page at http://www.openview.hp.com/products/nnm/index.asp, 2003.

[8] Bruce Lowekamp, David O'Hallaron, Thomas Gross, "Topology Discovery for Large Ethernet Networks," *ACM SIGCOMM Computer Communication Review*, 31(4):237-248, 2001.

[9] The MPI Forum, "The MPI-2: Extensions to the Message Passing Interface," Available at http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[10] MPICH - A Portable Implementation of MPI. http://www.mcs.anl.gov/mpi/mpich.

[11] Open MPI: Open Source High Performance Computing, http://www.open-mpi.org/.

[12] P. Patarasuk, A. Faraj, and X. Yuan, "Pipelined Broadcast on Ethernet Switched Clusters," *IEEE IPDPS*, April 2006.

[13] P. Patarasuk and X. Yuan, "Bandwidth Efficient All-reduce Operation on Tree Topologies," *IEEE IPDPS Workshop on High-Level Parallel Programming Models and Supportive Environments*, March 2007.

[14] P. Tan, M. Steinbach, and V. Kumar, “Introduction to Data Mining”, Addison-Wesley, 2006.

[15] Andrew Tanenbaum, “Computer Networks”, 4th Edition, 2004.

[16] IBM Tivoli. Web page at http://www.ibm.com/software/tivoli/products/netview, Feb. 2003.

BIOGRAPHICAL SKETCH

Joshua Lawrence obtained his BS degree in Computer Science from The Florida State University in Spring 2007. Under the advisement of Professor Xin Yuan, he will obtain his Master's Degree in Summer 2010 in Computer Science. Josh will begin the Ph.D. program in Computer Science starting Fall 2010. Josh's research interests are in high performance and parallel computing.