12-2008

A Novel Technique of Network Auditability with Managers In The Loop

Rian Shelley
Utah State University

A NOVEL TECHNIQUE OF NETWORK AUDITABILITY

WITH MANAGERS IN THE LOOP

by

Rian Shelley

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Robert F. Erbacher Major Professor	Stephen W. Clyde Committee Member
Chad Mano Committee Member	Byron R. Burham Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2008

ABSTRACT

A Novel Technique of Network Auditability

with Managers in the Loop

by

Rian Shelley, Master of Science

Utah State University, 2008

Major Professor: Robert Erbacher
Department: Computer Science

Network management requires a large amount of knowledge about the network. In particular, knowledge about used network addresses, access time, and topology is useful. In a network composed of managed devices, much of the data necessary can come from simple network management protocol (SNMP) queries. Other data can come from other databases, or analysis of existing data. In particular, layer-two network topology can be determined by analyzing the mac address forwarding tables of layer-two devices. The layer-two topology can be merged with a layer-three topology to generate a complete topology of the network. This information is useless unless it is easily accessible to the network manager; therefore, a simple interface should be used to give access to all of this data.

(57 pages)

CONTENTS

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

In any network, a manger must know what is happening, where it is happening, and when it is happening. It is necessary for the manager to be able to track down compromised boxes, and to be aware of his network topology.

1.1 Topology

In any network, proper management requires a good knowledge about topology [14]. For a small network, it is fairly easy to keep track of changes to the topology. This is also true for a large, well-planned, well-funded network. However, in some cases, because of the demographics of the users, it is difficult to keep track of the network topology, such as in the case of a university. Control of the network is difficult because of frequent tampering by experimenting professors and students. In this case, the layout of the network is not particularly important, as long as the managers understand it well.

1.2 Computer Management

Large networks can have many dynamically changing users. There are many solutions for keeping track of and authenticating these users, but in a flexible low-budget environment few of these solutions are available. Also, in the event of compromise or a network attack, it may be necessary to shut off network ports on layer-two and layer-three devices to prevent the spread of worms.

1.3 Overview

All of these problems can be solved by querying switches and routers for information. The trick is to cross-reference the information to make it useful. One can map layer-two devices using a variety of methods, such as the link layer discovery protocol (LLDP), MAC address forwarding tables, or by extracting the spanning tree used by the spanning tree protocol (STP). One can map the layer-three topology using trace routes, subnet locations, or route tables. The two topologies can be merged into a single map by replacing links between routers with sets of layer-two switches that are on the same subnet. It is easy

to keep track of users by querying the address resolution protocol (ARP) tables from the routers, and by querying the bridge MAC addresses from the switches. Cross referencing MAC addresses and IP addresses with registration information makes possible tracking down anomalies and users.

1.4 Purpose

The information that can be obtained about a network is to sort through. Given nothing more than an IP address and a timestamp, finding a compromised computer can be a huge job. It would be ideal to have a single place to make queries, that is, a place in to enter the item being searched for, and then (within reason), have the system return everything it could find out about the item in question. Thus, the initial user interface would start with nothing more than an input box and a submit button. It should also contain a minimum of dynamic content, so that web browsers could cache static information. That way, if the network were performing badly due to an outbreak or attack, very little information would be needed to update the screen.

The novelty of this solution lies in two areas. The first is the method for mapping layer-two devices. The layer-two map is merged into a layer-three map obtained using conventional methods. The second lies in the simple interface, and with the wealth of information that it can provide. The project, titled GULv3, a program that can track and display this information, but also display it in an easily usable manner.

CHAPTER 2
SOURCES OF INFORMATION

While this project pulls together information from many different sources, there are three primary methods for obtaining that information. The first is to assume that the information is readily available from a different source, usually a database of some sort. The second is to use some sort of query on the network device. The third is to process previously existing information to obtain new information.

2.1 The Physical Layer

For our purposes, we obtain information about the type of link (fiber optic, Ethernet, wireless, other) from the SNMP ifType management information base (MIB) value [20, 22]. We obtain the information about switch port to wall jack mappings from a database maintained by the people who installed the equipment. While useful, this information may not be necessary and is, therefore, not included in the initial version of the project.

2.2 The Data Link Layer

We obtain information about VLANs from SNMP queries of layer-two devices using the dot1qVlan subtree of the bridge extensions MIB [21]. Information about neighboring switches found by discovery protocols can be obtained via the IETF physical topology MIB [4, 13]. Information about the current spanning tree topology can be obtained using the dot1dStp group of the bridge MIB [21]. The MAC address forwarding table can also be obtained using the dot1dTpFdb table of the bridge MIB [21].

2.3 The Network Layer

IP routing paths, and, to a lesser degree, topology can be obtained using trace routes [37]. The ipAddressTable or the ipAddrTable of the IP MIB can be used to find the IP address and subnet mask associated with an interface [33]. Discovery methods, such as discovery protocols [4] and even internal routing protocols like OSPF [25] or ISIS [27] can reasonably identifier who their neighbors are. Routing tables are available via ipRouteTable in the IP MIB [33].

2.4 Other

There are other pieces of information useful to this project. Registered MAC addresses and contact information an be made readily available by access to some registration database. The first three bytes of a MAC address can be used to identify the manufacturer of a network card [3, 12]. Such identification can easily be defeated by spoofing the MAC address, but this is more of a management tool than a security tool. Used bandwidth can be obtained via the ifTable in the interfaces MIB [23], or from another source that collects it such as Cricket or MRTG [15, 26]. Flow information can be exported from some types of routers [19], or by using the FProbe software [9]. SFlow [28] packets can be exported from most managed networking devices. Most of these sources of information are useful, but they are outside of the initial scope of this project.

CHAPTER 3
LAYER-TWO MAPPING

Discovery protocols, the spanning tree, and the MAC forwarding table can be used to discover the topology of a layer-two network.

3.1 Discovery Protocols

Several discovery protocols are capable of finding nearby network devices, such as LLDP [4] or CDP [7]. However, most of them are either proprietary, or relatively new, and they do not interoperate with each other. Thus while useful, they may not be widespread enough to solely rely on them.

3.2 Spanning Tree

The spanning tree designated root field in the dot1dStp MIB [21] can help find the topology as well. Doing so is as simple as creating a list of edges between the current switch and the designated root, wherever the designated root is not the current switch. Such a maneuver has the advantage of also showing links that are shut down by STP. This will allow us to extract the real world graph from the switches, instead of the used spanning tree. If there is a mismanaged switch, it is still possible to find its MAC address if it happens to be the designated bridge of a link. However, in this case, the topology may be incomplete.

3.3 Mac Forwarding Table

If the MAC address forwarding table is available, a topology can be built from the bottom up using the following algorithm. We developed this algorithm so that there is a method of discovering topology when discovery protocols are not available.

Consider a switch N_1 as a node on the graph T. The switch has $p \cup u$ ports, where p is the set of ports that are connected to other switches, and u is the set of ports connected to users, or not used at all. $p \cup u$ is what is obtained when the switch is queried. We remove the set p from $p \cup u$ by obtaining the MAC addresses used by each switch. Then, we select only those ports that have forwarding entries for other network devices.

If a spanning tree is not established as the topology of the set of switches, then the

network is not functional because of broadcast storms [2]. Therefore we can assume that each forwarding entry follows a branch of a spanning tree. Each port in p knows the MAC addresses for all of the network devices that should be forwarded through it. The following is the port forwarding table for the network shown in Figure 3.1:

$$A = ((C), (B), (D, E, F, G))$$

$$B = (A, C, D, E, F, G)$$

$$C = (A, B, D, E, F, G)$$

$$D = ((A, B, C), (E), (F, G))$$

$$F = ((A, B, C, D, E), (G))$$

$$G = (A, B, C, D, E, F)$$

Figure 3.2 also illustrates this data. Using the MAC forwarding table, we can now write an algorithm to reconstruct the topology. Consider each node N, in the set T to be a tree with one element. Then:

1. Consider each N in T
2. Consider each port p_n in N
3. Consider each M in T where $M \neq N$
4. If the tree M contains all of the nodes in the direction of p_n, then remove M from T, and add M as a child of the tree N

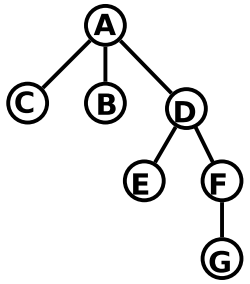

Figure 3.1. A simple switched network.

5. Repeat until no more changes are made to T, or T only contains one tree.

Thus, the example given in Figure 3.1 is solved in three passes. Figure 3.2 shows the initial setup, in which the circular nodes are trees and the rectangular nodes represent forwarding tables. Figure 3.3 is the first pass, which shows the leaf nodes being absorbed into the other trees. Figure 3.4 shows the second pass, which moves node A into the tree at node D. The last phase, shown in Figure 3.5, shows node F moved into the tree with the root D, and the algorithm terminates, because we have a single tree. The resulting tree is not the same tree as the original, but if it is considered as a graph, then both trees represent the same spanning tree.

It is possible to infer the existence and position of an unmanaged switch if this algorithm completes, and there are p_n's without children. A set S of N's in G that composes all of the nodes in the direction of p_n indicates at least one unmanaged switch connected to p_n that also connects to the trees in S. If we use the same topology as is shown in Figure 3.1 except we assume that we are not polling switch D, the completed algorithm is as shown in Figure 3.6, which implies the network shown in Figure 3.7

This algorithm follows the logical topology, not the physical connection of the switches, but if there are no VLANS, or only simple VLANs, the two spanning trees are the same.

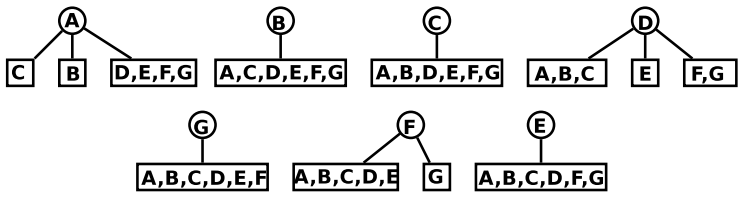

Figure 3.2. The initial data obtained by polling the MAC forwarding database on layer-two devices

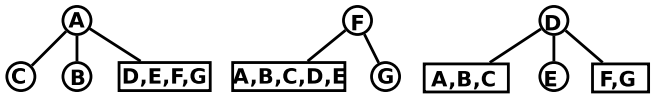

Figure 3.3. The first pass. All of the leaf nodes are absorbed into the other nodes.

This is the only method capable of producing a complete tree when missing key information. While the complete tree is not necessarily correct, its inconsistencies are predictable, and the information can be used in combination with other methods to produce a more accurate tree.

3.4 Combining All the Methods

Each algorithm produces the same tree on a well managed network. However, if a network has badly behaved or unmanaged devices, each algorithm produces slightly different trees. Consider the network in Figure 3.8. The MAC forwarding algorithm produces the tree in Figure 3.7. The presence of the missing node is inferred, but there is no knowledge of the redundant link. The spanning tree algorithm produces the tree in Figure 3.9. The tree is produced by using the designated root of each link to infer edges. This means that any known node knows about any link to a higher numbered node.

If we merge the two graphs by combining their lists of edges, and if we trust the spanning tree method wherein the edges conflict, the graph in Figure 3.8 is rebuilt without using any information from the unmanaged bridge.

In general, we can generate a topology using more than one method and combine the

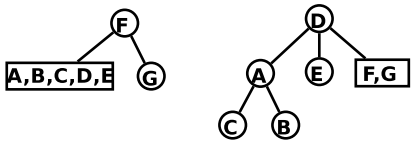

Figure 3.4. The second pass. Tree A gets placed into tree D.

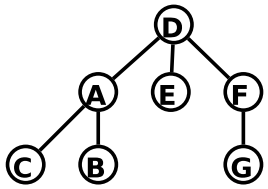

Figure 3.5. The last pass. Tree F gets placed into tree D. The original spanning tree has been recovered.

results to form a more accurate graph.

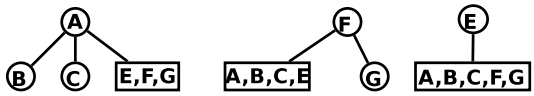

Figure 3.6. Without any knowledge of node D, this is as far as the algorithm can get. Because the forwarding tables of all three trees contain all the nodes from the other two trees, it is obvious that they are connected via an unknown bridge.

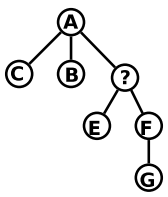

Figure 3.7. The tree inferred using the MAC forwarding based algorithm

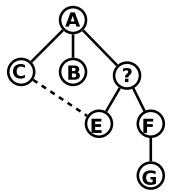

Figure 3.8. The node marked with ? is unmanaged, and the dotted line is a redundant link that has been shut down.

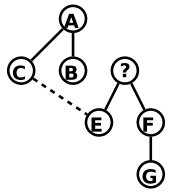

Figure 3.9. The tree inferred by extracting the spanning tree

CHAPTER 4
DEVELOPMENT

We developed this project on a live network with real data, with the target users being a heterogeneous group of security professionals, administrators, and service desk personnel. Additionally, we used a spiral model [16] to implement and refine the project. The requirements for each iteration of the spiral are included below. Finally, unit tests like the one shown in Figure 4 were used to verify the requirements and the stability of the code.

4.1 Initial Requirements

Initially, the only requirement was a mechanism that could be used to gather and query ARP information. We created this as a single interface that could accept an IP address or a MAC address.

4.2 More Data

The next iteration of the cycle included a requirement that the system also query MAC addresses from the forwarding tables of layer-two devices. Created as a separate interface, it could accept a layer-two device or a MAC address

4.3 Less Complication

Per a suggestion that the interface be simplified, we modified the requirements accordingly. Instead of multiple interfaces for different types of searches, we combined the interfaces into a single interface that displayed the results of both searches.

4.4 Even More Data

Searching more than one location for the same piece of data resulted in requests for more places to be searched. Thus, we included known registration information such as the registered IP address, MAC address, and any contact information. We also looked up and displayed the MAC Vendor code.

4.5 Cleanup and Presentation

The displayed data was too cluttered and disorganized to read easily necessitating a clear hierarchical display of the data. Our test runs also showed us that when a search did not yield a match, an alternative search should be included. For example, if a search on a hostname resulted in an IP address, an ARP search could yield a MAC address, which in turn could be applied to a search for the MAC address in the switch forwarding entries. We also tried highlight and describe obvious problems.

4.6 Topology

Frustration with the inability to keep track of topology resulted in including a mapping mechanism in the project. To accomplish this, the system first automatically created layer-two and layer-three and then merged into a cached map that could be recalled on demand. Any search for a network device now automatically displayed this topology.

4.7 Storage, Reliability, and Detail

Up to this point, the data was stored in flat files, which were completely loaded and searched every time the interface was used. Only a limited amount of precision and data was kept in order to keep search times reasonable. The project only kept track of the first and last occurrence of its ARP and MAC tables. ARP entries were only updated every 15 minutes. MAC tables every half an hour. The polling was adjusted to store its results in a database, keeping track of detailed intervals. The polling was set to occur as often as possible, and could poll multiple instances at a time.

4.8 Data Relationships

When there were conflicts in the relationships among the data, the presentation was confusing. For instance, a search for a hostname could yield a MAC address and an IP address. A search in the ARP database might return different MAC addresses and IP addresses than the current registration, in which case, the search would include the newly discovered addresses. It was difficult to tell the relationship of each search to the original search term. A subquery embedded in the original search allowed the user to search only

the terms s/he was interested in, and it clearly (see Figure 4.2) showed the relationship between the original search and the new search.

4.9 Current Use Cases

The following sections discuss use cases for which GULv3 has been designed to handle.

4.9.1 Looking Up Registration Information

The simplest use case is looking up registered information about a computer, and perhaps seeing if the registration varies from what is really happening. Searching for a computer by hostname, MAC address, or IP address returns contact information, and addresses associated with the search term.

4.9.2 Finding a Compromised Machine

Frequently, an organization's security group obtains information about a machine that may be compromised. Usually, all they have is an IP address and a timestamp. Searching GULv3 for the IP address either yields nothing, which tells us that the IP address was spoofed during the attack, or if the IP address is registered, it yields contact information and ARP information which tells us the MAC address being used during the time of the attack. Running a subsearch on the MAC address tells us exactly which switch port the computer was plugged into, and very likely also yields contact information.

4.9.3 Tracking Rogue Wireless Access Points

Currently, USU has a campus-wide 802.11b/g implementation. The managed wireless access points report the MAC address of any rogue wireless network on campus. Frequently these networks are broadcast using off-the-shelf wireless routers that also do NAT [34, 35]. These devices typically have a different MAC address on the network-facing side than on the client-facing side. GULv3 can search the first then reported by the campus AP. This finds any MAC address that matches the first 10 digits, and thus yields both MAC addresses used by the NAT device. Expanding the MAC address plugged directly into a switch returns any information known about that MAC, including what switch port it is plugged into.

4.9.4 Finding Evidence of ARP Poisoning

A malicious machine on a network may try to poison the ARP cache of the router [38]. Evidence of this shows up in GULv3. A targeted IP address shows up as having more than one MAC address. Over a period of time, a MAC address that is probing for data will show that it is using different IP addresses, as shown shown in Figure 4.3. Even if the MAC address is spoofed, it will show up in the switch forwarding table, and so can be traced to the switch port and possibly the wall jack.

4.9.5 Searching for Bottlenecks

If a network administrator wants to find the source of a bottleneck, s/he can find the switch attached to one end of his link, and use the mapping feature to follow it to the other end, or to the border of the network. Thus he knows which links are being used to transfer the data.

4.9.6 Verifying Network Topology

After new devices are added to the network an administrator can search for devices in GULv3 and use it to audit the network topology, thus allowing the administrator to verify the shape and make-up of the network.

4.9.7 Troubleshooting Network Connections

If a customer's connection to the network is not working properly, an administrator can quickly use GULv3 to determine if the switch is aware of the customer and if the customer is using a valid IP address. This allows the administrator to answer many of his/her own questions, rather than relying on the answers the customer, who is often an inexperienced user, is likely to provide.

```
class ArpTest(unittest.TestCase):
    def testWalk(self):
        self.CreateTable()
        self.arptable.poll(self.db)
        result = self.db.query("""
            select ip,mac,startstamp,stopstamp
            from arpentries
            where ip='%s';
            """%(self.gateway))
        self.assertTrue(result)
        startstamp = ""
        stopstamp=""
        for row in result:
            if row[0] == self.gateway:
                startstamp = row[2]
                stopstamp = row[3]
                break
        else:
            self.assertFalse(1)
        self.arptable.poll(self.db)
        result = self.db.query("""
            select ip,mac,startstamp,stopstamp
            from arpentries
            where ip='%s' and startstamp='%s';
            """%(self.gateway, startstamp))
        self.assertTrue(result)
        os.kill(self.snmpdpid, signal.SIGINT)
        self.assertFalse(self.arptable.poll(self.db))
        self.arptable.save(self.db)
        result = self.db.query("""
            select ip,mac,startstamp,stopstamp
            from arpentries
            where ip='%s' and startstamp='%s';
            """%(self.gateway, startstamp))
        self.assertTrue(result)
        self.assertNotEqual(result[0][3], stopstamp)
```

Figure 4.1. Part of the unit test for the ArpTable class

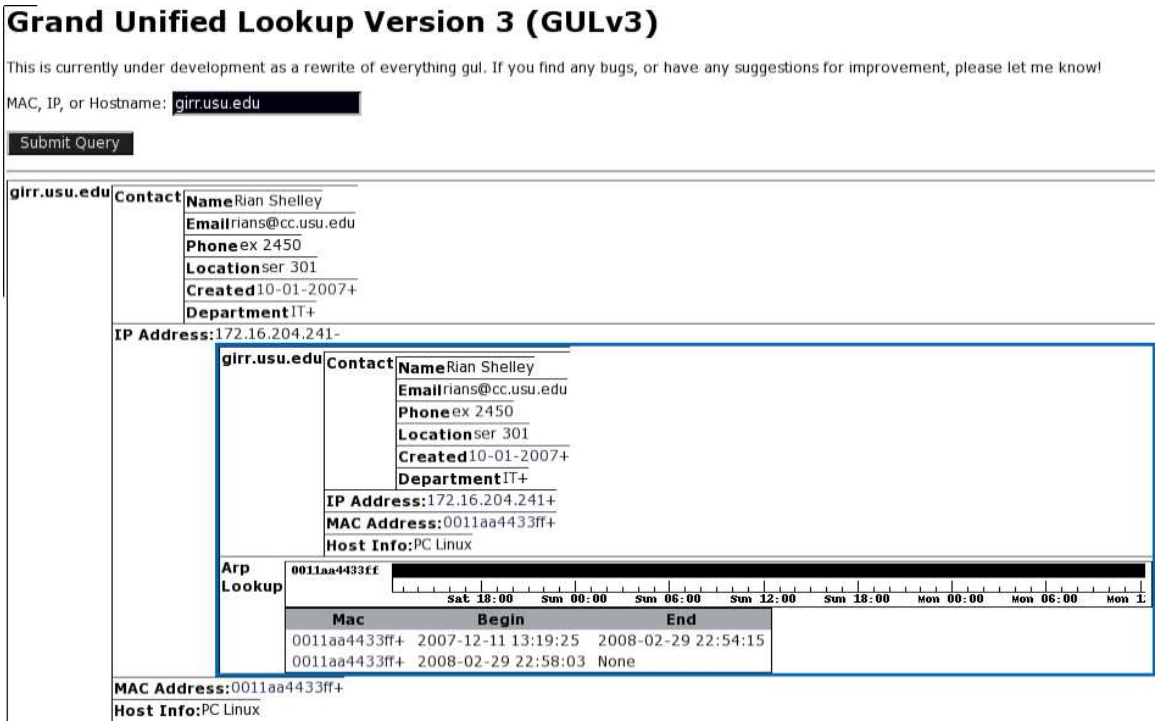

Figure 4.2. A host-based query, with a subquery on its IP address

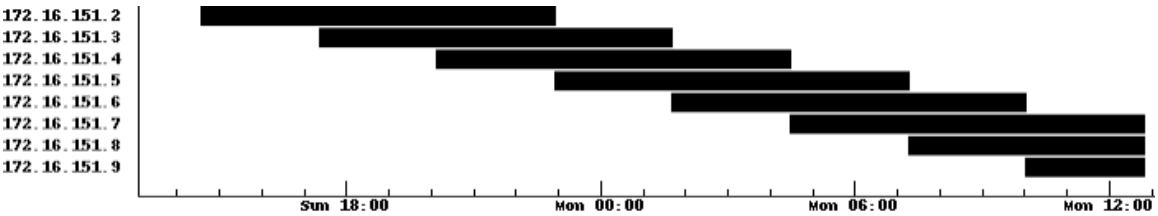

Figure 4.3. Evidence of ARP poisoning. A single MAC address has multiple IP addresses during overlapping time intervals.

CHAPTER 5
IMPLEMENTATION

This project contained no CPU-intensive tasks and lent itself to a convenient object-oriented interface. The large amount of data collected, stored, and searched, required a database. We chose Python as the language to implement this project because it met the criteria nicely, and supported the rapid development cycle necessary. PyUnit was used to implement unit tests. These were used to quickly verify code stability and to verify that the requirements were being met. In total, there are 4648 lines of code in 47 files.

5.1 Database

The database class contains all the searches and queries required by the rest of the code. It maintains a connection to the database backend during the lifetime of the object. Initially, the database class was created using PostgreSQL database [10]. After concerns about performance, the database class was separated into backend specific and backend independent code. A MySQL [8] backend class was created to compare with the PostgreSQL class as shown in Figure 5.1. At the time, the MySQL backend performed much better than the PostgreSQL backend, so the majority of the development since then has been on the MySQL code. Currently, tests show nearly equal performance between the two.

The database agnostic interface verifies the existence and configuration of the tables needed to store the data. If they are not present, it tries to create them. Additionally, it indexes any that are searched.

Initially, the database kept the list of open ARP entries and MAC entries in a separate

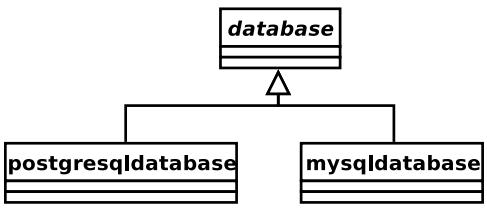

Figure 5.1. The database object

table from the closed intervals. However, this did not create a performance increase, so they were combined by allowing the stop time for an entry to be null.

5.2 Pollers

A large number of our data sources are based on SNMP. This lends itself to a polling paradigm, wherein multiple targets need periodic polling. The speed of the polls is bound by the network latency, so the ability to poll multiple devices at once increases performance. This works well even on a single processor. Each Poller creates its own database object. This yields two inherent classes, a Poller and an IPollable interface. The IPollable interface is implemented by classes that gather information on a periodic interface. Figure 5.2 shows the class diagram representing this relationship.

A Poller maintains a queue of IPollable objects. The object at the front of the queue is polled and placed at the back of the queue. New items may be inserted or removed from the queue at any time.

IPollable objects have a Poll() method and a minimum wait time. The minimum wait time is necessary to prevent a particular data source from being polled too often.

The driver program creates multiple Pollers, and distribute the IPollable objects among them. Each Poller object maintains its own thread and can execute in parallel. It is possible

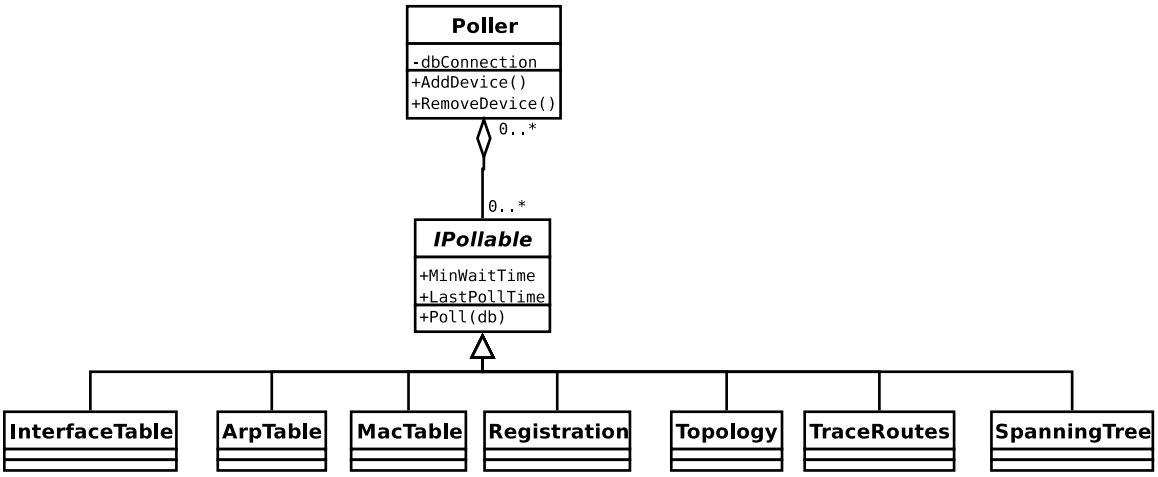

Figure 5.2. The Poller object, with the IPollable interface.

to place the Pollers on different machines if necessary. Each Poller also maintains its own database connection, which is provided to the IPollable object when its Poll() method is called. In this way, multiple machines using multiple processes can simultaneously poll several devices at once.

Figure 5.3 shows the thread that does the polling. The first if statement checks to see if we need to do any jobs when the queue cycles. Currently, it only contains a check to verify that the queue does not cycle too fast. This would be the case if the queue were empty or only contained a few items. The second if statement checks to see if the IPollable object needs to be removed from the queue. If not, the system polls the IPollable object, and then places it in the end of the queue.

Given the long lifetime of the Pollers, each one can optionally log everything it does to a file. This is necessary to be able to track down bugs that only occur after a significant amount of running time. The logging code can optionally email exceptions to an email

```
def __threadPoller(self):
    pollstart = time.clock()
    self.counter = len(self.queue)
    while self.running:
        if (self.counter <= 0):
            now = time.clock()
            if now < pollstart + MINELAPSED:
                time.sleep(MINELAPSED)
            self.counter = len(self.queue)
            pollstart = time.clock()
        if len(self.queue) > 0:
            if (len(self.removable) > 0 and
                    self.removable[0] == self.queue[0]):
                self.removable.pop(0)
                self.queue.pop(0)
                self.counter -= 1
                continue
            self.queue[0].poll(self.db)
            self.queue.append(self.queue.pop(0))
        self.counter -= 1
```

Figure 5.3. The polling thread.

address. We added this feature in order to track down several bugs involving broken devices that returned inconsistent data.

5.3 GenericGenerator

There were many IPollable objects that had to be created and placed into Pollers. We created the GenericGenerator class to ease the loading of these objects. Figure 5.4 contains its code. It takes a class object and a filename, and reads a colon separated list of constructor arguments for each new object of the class from the file.

5.4 SNMPQuerier

Many of the objects that inherit the IPollable interface use SNMP [18] to gather information. An SNMPQuerier class associates a UDP port, an SNMP OID, and a community string together. We use the resulting object to query a target. A callback function in the calling object returns the results of an SNMP walk. We initially implemented the SNMPQuerier object using the pysnmp [11] library, but its performance was not sufficient. A second SNMPQuerier class is currently used, which uses piped command line output from

```
class GenericGenerator:
    def __init__(self, IPollableClass, filename):
        self.IPollableClass = IPollableClass
        self.filename = filename
    def GetPollables(self, start=None, stop=None):
        file = open(self.filename)
        newclasses = []
        for line in file:
            if (len(line) == 0 or line[0] == "#"):
                continue
            args = line.rstrip().split(":")
            if (start != None and stop != None):
                args = args[start:stop]
            newclasses.append(self.IPollableClass( *args ))
        return newclasses
```

Figure 5.4. The GenericGenerator class

utilities in the Net-SNMP suite [5].

5.5 DeviceInterfaces

DeviceInterfaces implements the IPollable Interface. It queries ifTable from the interfaces group of MIB-II [22] for the interface MAC address and description, the sysServices value from the system group of MIB-II [29], the ipAddrTable from the IP group of MIB-II [33] for the IP addresses and subnet masks present on an interface, and the ifAlias value from the ifXTable of the ifMIB group of MIB-II [22].

The IP address is not available for an interface which is solely layer-two, so the sysServices value is used to differentiate between layer-two and layer-three devices. The Database object, provided by the Poller, shores these values in the database. This makes a list of every interface on every device available, along with any IP addresses or descriptions configured on the device.

The information provided by this class only changes when new devices are added or new subnets configured. This information may not change very often, so the minimum wait time between polls can be fairly large. The default is set to 24 hours.

5.6 ArpTable

The ArpTable class models objects that are associated with a single device. It polls the ipNetToMediaPhysAddress value from the ipNetToMediaTable of the IP group of MIB-II. This gives us all the ARP entries on a layer-three device. A list of all the ARP entries discovered during the previous poll is obtained from the database. If the ARP entry still exists, it is left unchanged. If it no longer exists, an ending timestamp is associated with the ARP entry. Thus, the ArpTable maintains a list of ARP entries associated with starting and ending times for when they were seen. For the sake of convenience, it also maintains a table that merely records the last time an ARP entry was seen.

ARP entries change frequently, so each device should be polled as often as it can. No minimum wait time is associated by default with this class.

Figure 5.6 shows the code for the ArpTable class. The initialization creates the SNMP poller and saves the device. The poll uses the SNMP poller to walk the SNMP tree for the

ARP table and saves the data. The class implements the eachwalk() function required by the SNMP poller which uses it as a callback. The save loads all the existing ARP entries, and marks the ones that are not part of the new set polled as closed. The remaining entries from the new set are added to the database.

The ArpTable was the first class we implemented for GULv3. Its simple design has left it relatively unchanged throughout the development cycle. Figure 4 shows part of the unit test associated with the ARP table, although some of the setup and cleanup code is omitted. This class creates a temporary SNMP server, and queries the routing tables of the tester to get an idea of what data should be obtained. It then uses that to get the results of

```
class ArpTable(IPollable):
    "Query and store an arp table from a device"
    def __init__(self, device, port=161):
        self.device = device
        self.poller = SNMPPoller(
            (1,3,6,1,2,1,4,22,1,2), "snmpcommunity", port)
        IPollable.__init__(self, device)
        self.iptomac = {}
    def poll(self, dbconn):
        if (self.poller.walk(self.device, self)):
            self.save(dbconn)
            return True
        return False
    def save(self, db):
        for arp in db.getcurrentarps(self.device):
            if ((not self.iptomac.has_key(arp[0])) or
                    self.iptomac[arp[0]] != arp[1]):
                db.stoparp(arp[0], arp[1], self.device)
        for ip in self.iptomac.keys():
            db.savearp(ip, self.iptomac[ip], self.device)
        del self.iptomac
        self.iptomac = {}
    def eachwalk(self, name, object):
        mac = object.split(":")
        self.iptomac["%d.%d.%d.%d" % name[-4:]]=
            "%02x%02x%02x%02x%02x%02x" % [int(d, 16) for d in mac]
```

Figure 5.5. The ArpTable class. This is a typical IPollable object.

the test. We eliminated many painful bugs because of the unit testing done in this project.

5.7 MacTable

The MacTable class looks up MAC address forwarding tables for a specific device. This essentially associates a MAC address with a port. It links the MAC address to a device port discovered by the InterfaceTable class.

Not all of this information is blindly committed to the database. Theoretically each switch in a layer-two broadcast domain is aware of each MAC address in the same domain. The class checks to see if a MAC address seen on a port belongs to a different managed device. If it is, this implies that the port is not directly connected to a user. This information is redundant and not useful for locating computers, as they will already have an entry on the port they are physically plugged into. However, the MAC address of the remote device is useful in determining the layer-two topology as described in Section 3.3. Therefore unknown MAC addresses on a port pointing to another device are discarded, but known device MAC addresses are stored in a separate table to be processed for topology information.

Any information on a port that does not appear to point to another managed device is then committed to the database. The class maintains start and stop times for each entry the same way as is maintained by the ArpTable class.

Mac forwarding information changes as frequently as ARP information does. This class similarly has no minimum wait time.

5.8 Registration

The Registration class inherits the IPollable interface. It is different from InterfaceTable and ArpTable in that it does not obtain its information via SNMP but from a separate database maintained by network administrators. The current implementation reads the information from a complex flat file. Pyparsing [24] is used to parse the file for useful information. In tests this usually takes several seconds so the information is cached in the database for rapid searching. Initially, buggy information was obtained using a line-by-line scanner that queried the file directly, but profiling revealed that 95% of the search time was spent parsing the file.

At USU, automated updates to registration happen every hour, although manual updates can happen at any time. Most of the information never changes. Therefore, we set the default minimum wait time to an hour.

5.9 TraceRoutes

We did not implement the TraceRoutes class in the current project. However, we include it here because it was implemented in previous versions of GUL. It uses output from the traceroute [37] command to obtain information about layer-three logical topology. It is somewhat useful because it can obtain information from layer-three devices that are not managed. However, from the perspective of a single Poller, it is unlikely that it will find redundant links, since the paths for packets will tend to be the same.

5.10 SpanningTree

Similarly, we did not implement the SpanningTree class. However, it has immense benefits, especially if it is combined with other mapping methods as described in Section 3.2 and Section 3.4.

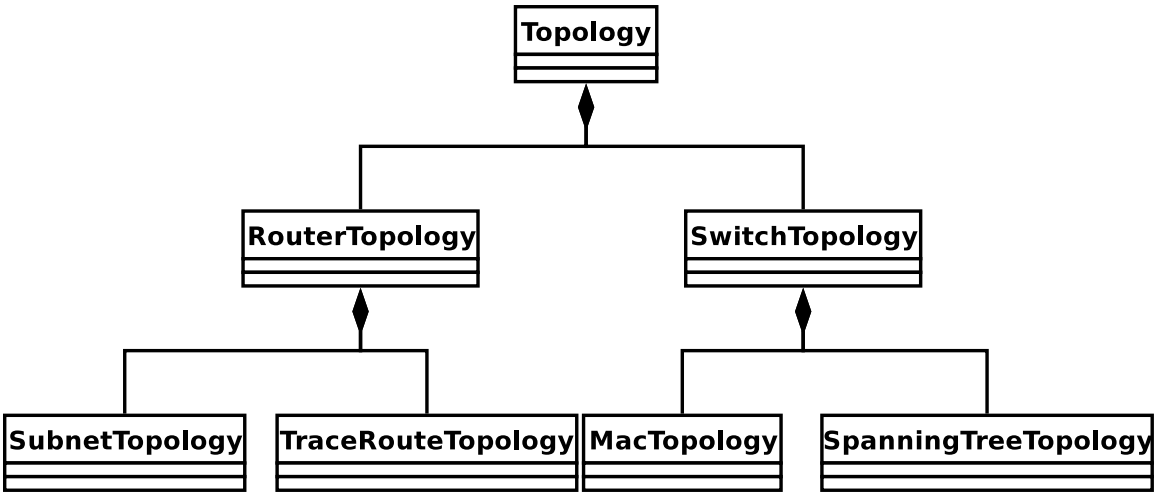

Figure 5.6. The composition of the Topology class

5.11 Topology

The Topology class implements the IPollable interface. Its functionality is divided up as shown in Figure 5.6. A Topology object combines the edges provided by RouterTopology and the edges provided by SwitchTopology. If an edge contained in RouterTopology connects two nodes, A and B, and there exists a graph G in SwitchTopology that contains both A and B, it is assumed that the L3 edge AB is a layer-three representation of the graph G. Thus, AB is removed, and G is used instead. This corresponds to the case wherein two routers are not connected directly but have a network of switches between them.

5.12 SwitchTopology

SwitchTopology is a class that combines the data represented by MacTopology and SpanningTree as described in Section 3.4. It includes as many different types of layer-two mapping algorithms together as necessary to produce an accurate graph.

5.12.1 MacTopology

MacTopology is a class that implements the MAC address forwarding table mapping algorithm as described in Section 3.3. It uses the data produced by the MacTable objects for this purpose. Figure 5.7 shows the initialization code. This code translates the MAC addresses into the nodes they represent, and sorts them by category. A MAC that represents a router goes into a special case, so that we can attach the resulting network into a layer-three map. A MAC that exists on its own device is discarded. This happens when somebody

```
for (portid, mac) in db.topologyinfo():
    if (mac in routermacs):
        nodes[porttodevice[portid]].routers[portid] = routermacs[mac]
    elif (mac in mactodevice and portid == mactodevice[mac]):
        pass
    elif (mac in mactodevice):
        nodes[porttodevice[portid]].addrelation(
            portid, mactodevice[mac])
```

Figure 5.7. This loop sorts the information gleaned by the MAC addresses.

plugs a switch into itself and it confuses the algorithm. The rest are listed as "relations" of the node, since we don't know what the exact relationship is.

Figure 5.8 shows the code used to find a node whose descendants match the given set. Currently, this is done using a sequential search, but this code could be optimized by using a hash table or something similar to find a match. If the algorithm finds the given node, the system removes it from the list and returns it. Figure 5.9 shows the main part of the algorithm. It looks through all the node ports searching for a matching subtree for the set. If the algorithm finds said matching subtree, it adds it as a subtree of the current node.

```
def findsubtree(siblings):
    for subtree in list(nodes.itervalues()):
        if siblings == subtree.getdescendants():
            del nodes[subtree.id]
            return subtree
    return None
```

Figure 5.8. The findsubtree function. This locates a subtree whose descendants match the given set.

```
done = False
while not done:
    done = True
    for sw in set(nodes.itervalues()):
        if not sw in nodes.values():
            continue
        for port in set(sw.ports()):
            if not port in sw.ports():
                continue
            if not port in sw.routers:
                subtree = findsubtree(sw.getrelations(port))
                if (subtree):
                    sw.childrenlist[port] = subtree
                    if (sw.id in subtree.idtoport):
                        subtree.uplinkport = subtree.idtoport[sw.id]
                    done = False
```

Figure 5.9. The main part of the layer-two MAC mapping algorithm

The process is continued as long as there is a change made to the tree. When we consider a port, we immediately verify that it is still valid, since the loop can change the ports we are considering.

This complexity analysis is done assuming that n is the number of vertices, m is the average degree of its vertices, and o is the average number of vertices per disconnected subgraph. The code in Figure 5.8 has a complexity of O(n), due to its linear search. The code in Figure 5.9 is more complex to analyze. Each pass through the outer loop removes an arbitrary number of nodes from the list. The best case (which is unlikely) is that all of the switches are ordered so that every switch analyzed is a child of a switch yet to be analyzed. The average case is that half of the switches analyzed are eliminated on each pass. The worst case is that only one switch is removed each time. Therefore the best case is O(1), the average case is O($log_m(n)$), and the worst case is O(n). Looping through the nodes is obviously O(n), and looping through the ports is O(m). The sw.getrelations() call is O(o). All the rest of the operations are either reference assignments, which are O(1), or are sequentially executed with a bigger operation that eclipses them. Thus the total worst case complexity is O($n * n * m * (o + n)$), which reduces to O($n^2 * m * (o + n)$). The average case is O($log_m(n) * n * m * (o + n)$). In real world examples, m and o tend to be very small compared to n. USU's values at the time of writing were $n \approx 708$, $m \approx 3$, $o \approx 3$. o and m tend to stay fairly low, in order to keep network manageable. Thus the worst case reduces to O(n^3), the average case O($n^2 log(n)$), and the best case O(n).

5.12.2 SpanningTreeTopology

SpanningTree is a class that implements the spanning tree mapping algorithm as described in Section 3.2. Although currently not implemented, it would process the data saved by the SpanningTree objects.

5.13 RouterTopology

The RouterTopology class combines the results of TraceRouteTopology and SubnetTopology. The edges produced by the two objects are combined together to form a single graph.

5.13.1 SubnetTopology

The SubnetTopology class can generate a graph by assuming that any two interfaces in the data produced by the InterfacePoller objects that reside on the same subnet represent an edge. There are cases wherein this is not necessarily true, but the data is corrected later. If there happen to be more than two routers on the same subnet, and there is not a layer-two map that shows the relationship between them, we infer the existence of an unmanaged bridge.

5.13.2 TraceRouteTopology

The TraceRouteTopology passes through the list of edges obtained from the TraceRoute object. This also is not currently implemented, because of the higher accuracy from other methods.

CHAPTER 6
THE USER INTERFACE

The goal of the user interface portion of this project was to gather all these sources of information into a convenient interface that is simple and easy to use.

6.1 The Initial Page

The interface is very simple, as recommended by Raskin and Tidwell [31, 36]. It initially displays textbox and a submit button, as shown in Figure 6.1.

6.2 A Query

When a user makes a query on a hostname, the results will are shown in Figure 6.2. The information that is directly known about the hostname entered is displayed in a hierarchical manner. Most of the information is shown as a label, which is bold, and the value or values associated with it. The values are separated using lines in a manner that outlines the relationships among the different pieces of data. For instance, the logical entity girr.usu.edu contains contact information, an IP address, a MAC address, and an HINFO record. The contact information also contains a list of entries similarly displayed.

6.3 A Subquery

In a query, some of the data displayed is also searchable. For example, the IP address shown in Figure 6.2 is also a term that may be searched. The searchable elements are shown as links to the browser. They also have a plus sign next to them. The plus sign is used to display a subquery, as shown in Figure 4.2. The plus sign changed into a minus sign,

Figure 6.1. The initial layout

indicates that the user can use it to remove the subquery from the screen if desired. The purpose of the subquery mechanism is to allow us to see how the new data relates to the old data. The entire subquery is outlined in a separate color, so that it can be easily seen as separate from the rest of the data. The data in the subquery itself is basically identical to the initial query, but it is included because the system does not determine whether or not the data is redundant. There is no HINFO record associated with the IP address, but it does have ARP entries, which are not associated with a hostname. The data is shown in two formats. The first is a graph where the X axis is time, and the Y axis is the MAC address using the IP address during that time. In this example, the MAC address was constantly associated with this IP for the entire time span shown by the graph. The second format is a list of time intervals, which is shown for completeness. More interesting examples of this time graph are shown in Figure 6.3 and in Figure 4.3.

Clicking on a MAC address searches the database from the perspective of the MAC address as shown in Figure 6.4. The ARP lookup is now a list of IP addresses, rather than a list of MAC addresses. There is also a MAC lookup section that contains the location of

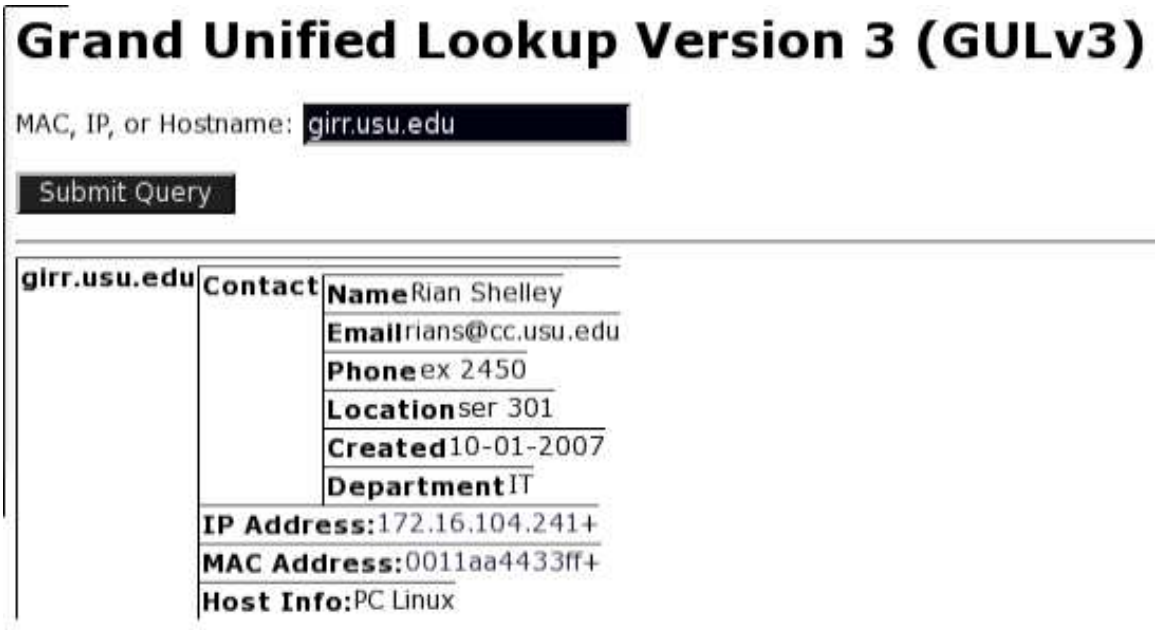

Figure 6.2. A host-based query

switches where the MAC address has been seen.

6.4 Network Devices

Clicking on a network device or entering it into the search field gives us a slightly different view. It shows a list of ports and a network map similar to the one shown in Figure 6.5. The map contains the devices, the connections between them, and the port numbers for the connections. The searched device shows up as a different color than the rest, and serves as the root of the displayed tree. The exception is that the immediate parent of the searched node is also shown. We may click on any of the nodes on the map, which runs a search on that particular node. In this way, we have the ability to traverse

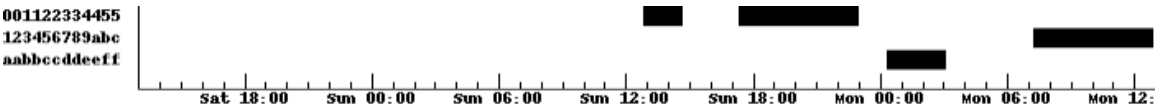

Figure 6.3. A dynamic IP, used by different MAC addresses during different intervals

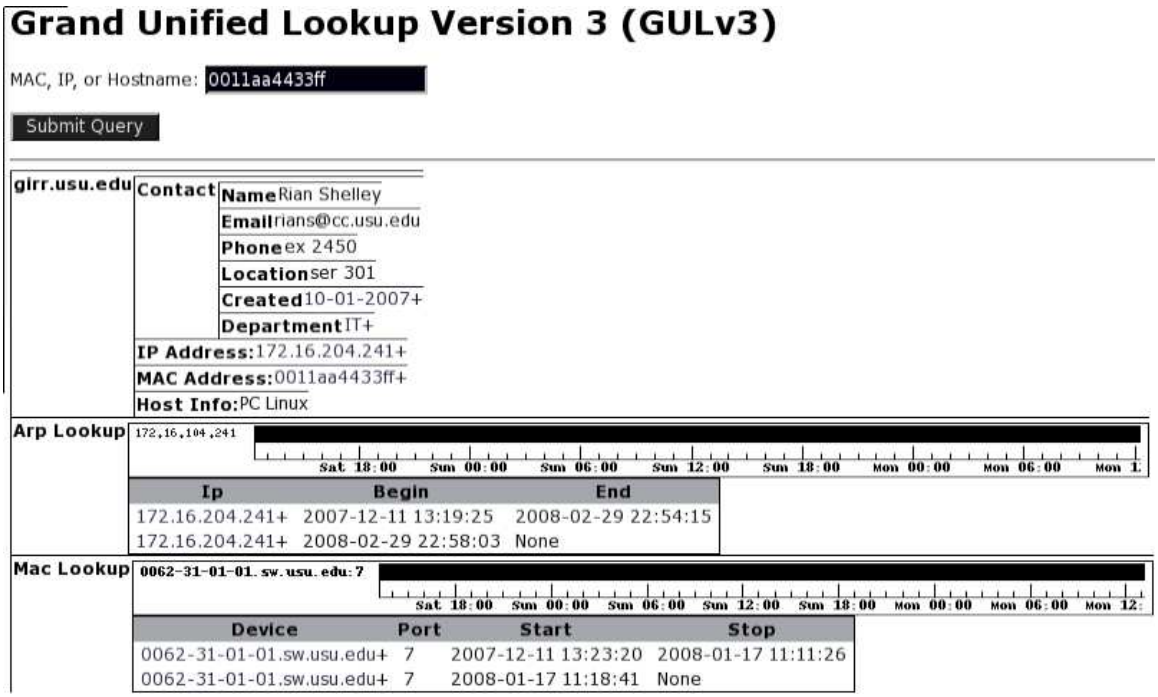

Figure 6.4. A search for a MAC address

the topology of the network entirely by clicking on the nodes on the map. The current implementation maps routers, switches, and wireless access points. Internally, the wireless access points are treated like switches.

Figure 6.5. A localized map of the network. This example happens to contain a router, two switches, and two wireless access points.

CHAPTER 7
IMPLEMENTATION OF THE USER INTERFACE

As described, the user interface breaks down into two types of classes for which there are natural interfaces: the ISearch interface, and the IDrawable interface. The main page creates any number of ISearch objects that produce a tree of IDrawable objects to represent the results. The IDrawable objects are displayed as HTML for the user. This relationship is shown in Figure 7.1

7.1 IDrawable

The IDrawable interface represents a class that contains an ordering, and can be represented as HTML [30] or XML [17].

7.1.1 Basic

Basic implements the IDrawable interface. It represents a simple string value. This class also checks the value to see if it is searchable. If it is, and it renders HTML, it includes

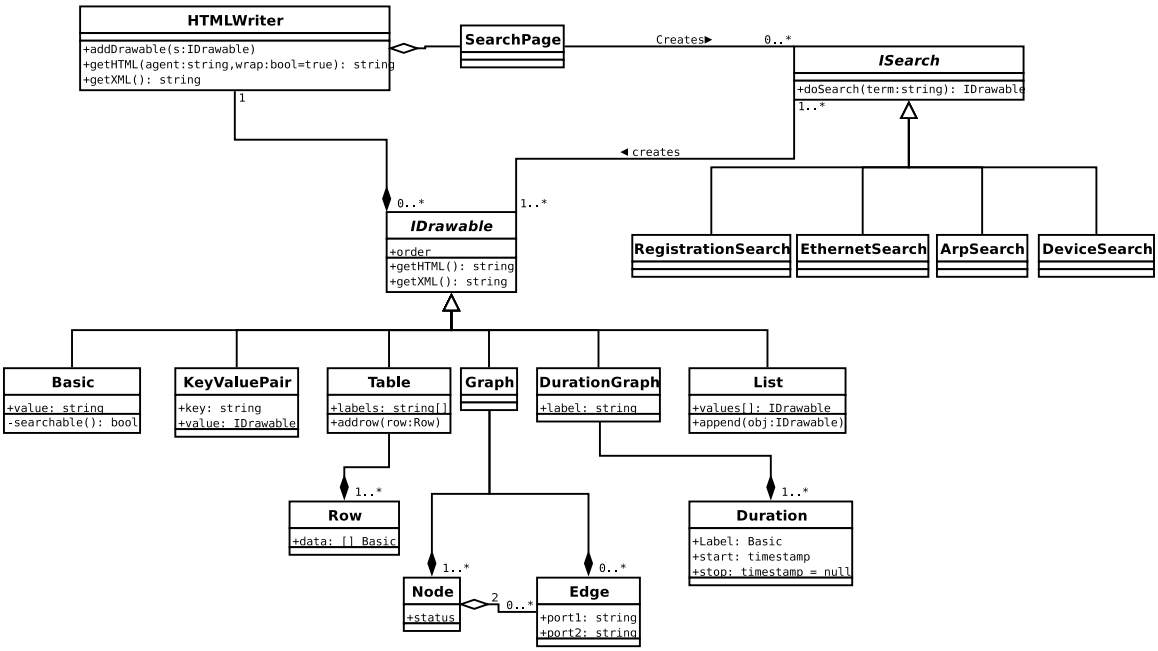

Figure 7.1. A diagram showing the relationship between the search and the results

a link so that the user can search by the term it contains.

7.1.2 KeyValuePair

KeyValuePair implements the IDrawable interface. It represents a key and a value. The key is a string and the value can be any IDrawable object. It is implemented in HTML by using divs and a cascading stylesheet to show the relationship between them. This is one of the two IDrawable classes that contain other IDrawable objects.

7.1.3 Table

Table implements the IDrawable interface. It displays information in a tabular format. It contains an array of strings that represent the column labels and a two-dimensional array of Basic objects. It is implemented in HTML as a table.

7.1.4 Graph

Graph implements the IDrawable interface. Graph represents a list of nodes and edges. It is currently partially implemented and uses the dot utility from graphviz [6] to produce a PNG [32] image. It outputs an HTML img tag that directs the browser to retrieve the produced image via a helper script.

7.1.5 DurationGraph

DurationGraph implements the IDrawable interface. It is designed to show periods of time. The current implementation outputs a PNG [32] image similar to that shown in Figure 4.3. It outputs an HTML img tag that uses the same mechanism used by the Graph class to retrieve a produced image.

7.1.6 List

List implements the IDrawable interface. It organizes a list of IDrawable objects. It is implemented in HTML as a ul [30] tag. This is one of the two IDrawable classes that can contain other IDrawable objects

7.2 ISearch

The ISearch interface describes an object with a doSearch() method that can return an IDrawable object. If the type of search is not applicable, it returns None.

7.2.1 RegistrationSearch

RegistrationSearch is an ISearch class that associates search terms with registered information that is gathered by the Registration poller. Typically, this returns a List of KeyValuePairs as shown as rendered HTML in Figure 6.2.

7.2.2 EthernetSearch

EthernetSearch is an ISearch class that searches for MAC addresses in the data collected by the MacTable class. It generates both a DurationGraph and a Table representing the data, encapsulated by a List. An example of this is the Mac Lookup section in Figure 6.4.

7.2.3 ArpSearch

ArpSearch is an ISearch class that searches for MAC or IP addresses in the data collected by the ArpTable class. It generates a DurationGraph and a Table encapsulated by a List, similar to the EthernetSearch object. An example is the ARP Lookup section in Figure 6.4.

7.2.4 DeviceSearch

DeviceSearch is an ISearch class that searches for MAC addresses, IP addresses, or hostnames that belong to managed devices monitored by GULv3. It formats each port associated with the device as a List of KeyValuePairs and encapsulates them as a List. It also displays a Graph if it can find topology information associated with the device. An example of the results is shown in Figure 7.2

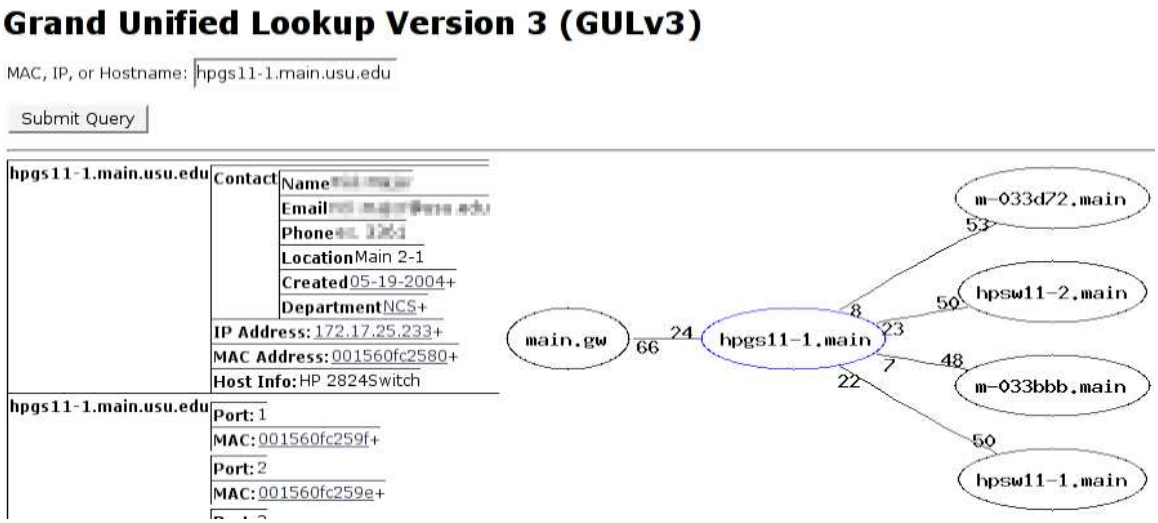

Figure 7.2. A search against the root switch of the CS department at USU. There are three switches and two wireless access points on this map.

CHAPTER 8
REAL WORLD TESTING

As described, we implemented GULv3 at Utah State University. We used a test phase to find bugs and to make improvements. In these tests, network administrators and service desk personnel with access to GULv3 used the system and provided feedback for its improvement. The users are Utah State University employees who have signed a privacy agreement as part of their positions. We used a subjective test was used because there has not been a similar tool available prior to the initial inception of GUL. We built a mechanism into the webpage that allows users to easily and anonymously submit suggestions for improvement and comments. Indeed, we implemented many of these suggestions as part of the project.

8.1 Suggestions

A number of suggestions were made for small improvements to the project. There was also some positive feedback.

The first suggestion was to change the displayed order of ARP and MAC entries. Previously, there was no ordering, which meant entries were displayed in the order they were created in the database. This was modified so that the most recent entries were displayed first.

The next suggestion was to change the font to make it more readable and to separate the elements more. In response, some padding was added to various elements to better separate them. The font size was increased slightly to increase readability.

Another suggestion was the ability to search for partial IP and MAC addresses. This was implemented using the SQL query like the one shown in Figure 8.1. The result of such a search is shown in Figure 8.2.

A fairly serious bug was noted, in which old data was being preferred over newer data. It turned out that many searches assumed that there would be only one result, and so the system used the result of the first row. The database displayed records in the order that they were created, which meant that the oldest record was always on top. The search was

modified to do a reverse sort on the timestamp field. Many other searches appeared to be vulnerable to a similar bug, so they were modified to either do the sort, or to only consider records newer than a default interval of 24 hours.

Another bug was noted wherein it was possible to do an SQL injection. We created a SearchTerm class to classify and sanitize the search term, which is accessible by using the term member variable. The sanitized version of the variable was not being used during a search for ARP information. That bug has also been fixed.

Pages that displayed network maps were noted to be slow. Some investigation revealed an O(n^2) operation occurring each time a map was drawn, where n was around 1500. One of the n terms was a linear search through an array. This term was removed by using a set to contain the data instead of an array, which does searches in O(1) time. The webpage

```
select
    mac,
    count(mac),
    device
from
    macentries
where
    mac like "001aA0%"
group by mac
order by count(mac) desc
limit 20;
```

Figure 8.1. SQL query used to find partial matches

Figure 8.2. The results of a search for a partial MAC address

loads very quickly now.

Another submission suggested that the ability to search for registrations by the contact information would be useful. This feature fit neatly into the PartialSearch class, and works well. It returns results as shown in Figure 8.3.

8.2 Performance

The test machine used to run the pollers and the database has a hyperthreaded [1] 3GHz Intel Xeon processor, with 4GB of RAM.

Figures 8.5 and 8.6 show CPU use of the database and poller over time. The processes tend to use the CPU in bursts. The reason for this is seen by examining the code in Figure 5.6. This class is a typical pollable object in that it first polls the information and then saves it as two separate states. The polling itself is network bound, waiting on the

Grand Unified Lookup Version 3 (GULv3)

MAC, IP, or Hostname: Shelley

Submit Query

Matching Registrations

Hostname	Name	Email	Phone	Location	Date
argh.ncs.usu.edu +	Rian shelley	rians@cc.usu.edu	ex. 8596	SER 321	01-05-2007+
audit1.ncs.usu.edu+	Rian Shelley	rians@cc.usu.edu	ex 2450	ser 301	10-01-2007+
audit2.ncs.usu.edu+	Rian Shelley	rians@cc.usu.edu	ex 2450	ser 301	10-01-2007+
audit3.ncs.usu.edu+	Rian Shelley	rians@cc.usu.edu	ex 2450	ser 301	10-01-2007+

Figure 8.3. The results of a search for a partial name

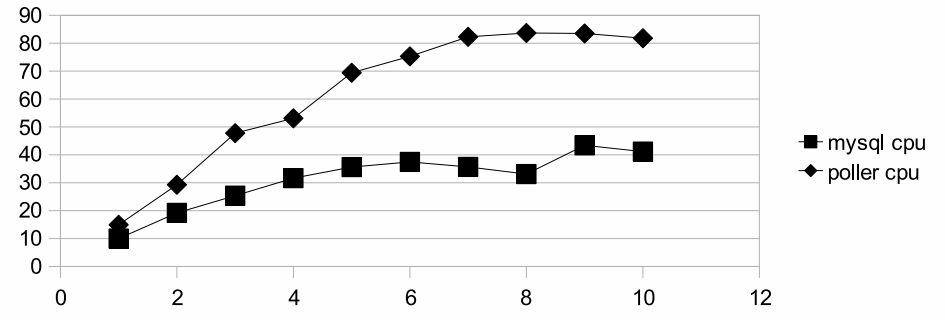

Figure 8.4. CPU usage for the frontend and database processes

remote device to respond, and uses very little CPU time. The saving has to process the data, and uses the CPU for a short period of time. For the most part, the CPU usage stays low, although it can spike if a particularly busy switch gets polled. The average CPU usage over a period of 479 seconds was 14.91%.

Figure 8.4 compares the average CPU use of the pollers and the database backend with how many simultaneous pollers are running. Because of the simple code, with only one poller, the CPU usage stays relatively low. As the number of pollers increase, the CPU usage increases linearly until about five pollers. After this, the sum of the CPU usage hits 100%. Because of the hyperthreading, some increase past 100% is still possible, but after about seven pollers, it more or less levels out.

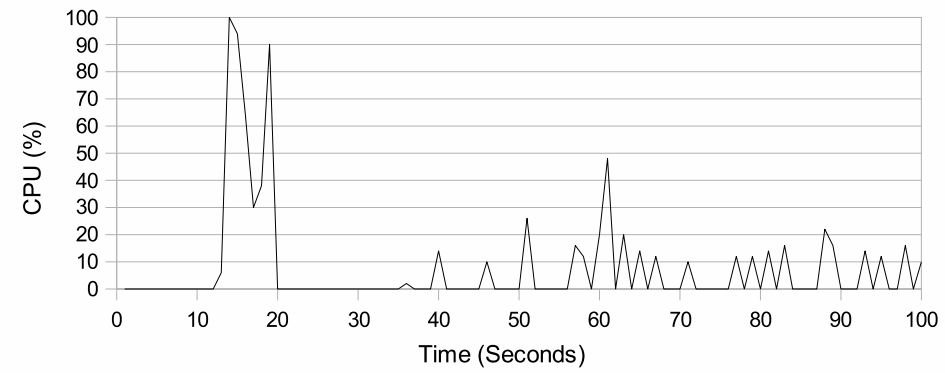

Figure 8.5. CPU usage for the database process

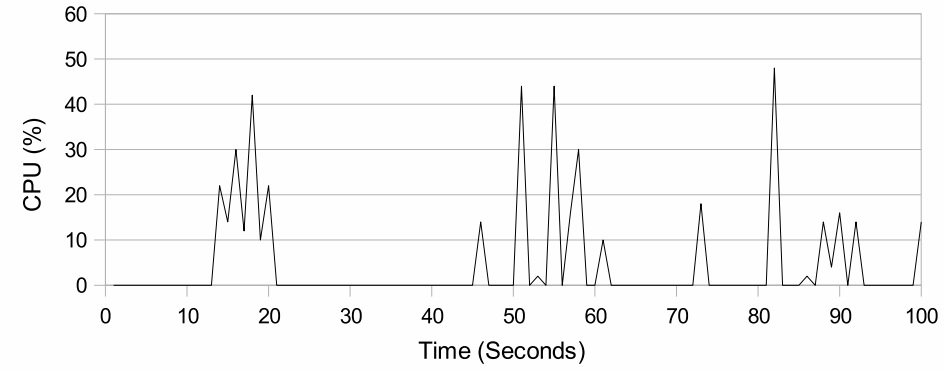

Figure 8.6. CPU usage for a single poller.

Figure 8.7 shows the relationship between the targets that can be processed per second and the CPU usage. The relation is more or less linear until the system is strained.

Figure 8.8 shows the disk usage compared to the number of pollers. The reads are apparently non-existent, which can be explained by the disk caching done by the operating system. The writes show the data that are committed to the disk after each poll. The data writes level off with the same number of pollers the CPU usage does in Figure 8.4

8.3 Usage

To get an idea of how useful users considered GULv3, the web page logs tracked how many requests per week were processed by GULv3; see Figure 8.9. A week-long period per data point was chosen to minimize the effect of scheduling differences between employees,

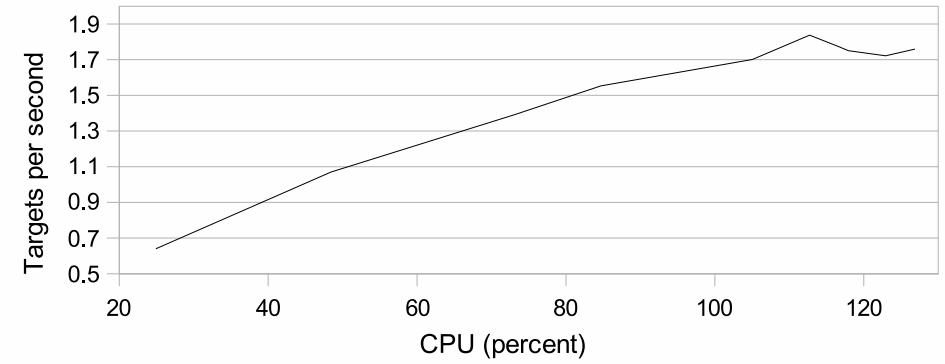

Figure 8.7. Targets per second related to the CPU percentage

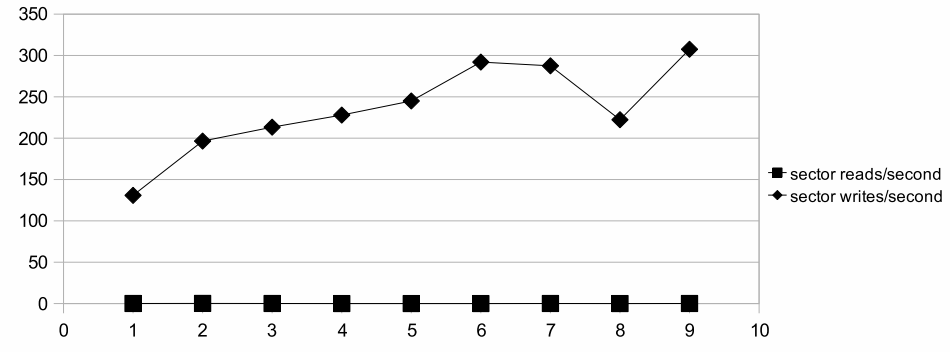

Figure 8.8. The disk usage compared to the number of pollers

and the lack of use on weekends. The IP address wherein most of the code changes were made was filtered out of the results. The trial period began in week 10, and the partial search feature was implemented in week 11.

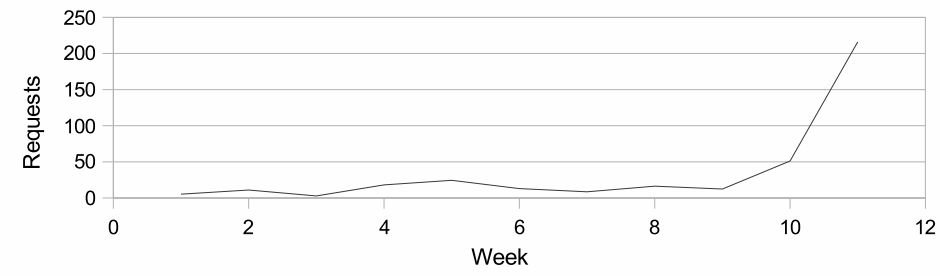

Figure 8.9. The number of requests per week over the lifetime of GULv3

CHAPTER 9
FUTURE IMPROVEMENTS

Much of the implementation of this project was a response to a specific need, or was a unique solution to a problem. A much more useful project would include more sources of information and combine them in meaningful ways.

It would be beneficial to include support for more types of network devices that support SNMP, such as wireless access points, printers, servers, etc. Using vendor-specific MIBs for devices would also be an improvement.

Much of the topology code included in this project is used to discover information that is readily available via LLDP. LLDP would also reveal more information about unknown nodes, and could provide a means of automatically discovering network devices, rather than using a network manager provided list. However, as described in Section 3.4, the most benefit would be derived by combining the two methods together to cover up their different shortcomings.

In a couple of places, the project currently uses deprecated MIB values. For instance, the ipNetToMediaTable value used in Section 5.6 only returns IPv4 addresses, and therefore is deprecated [33]. The ipNetToPhysicalTable should be used instead. On a similar thread, the project only supports IPv4 addresses. It could be easily updated to support IPv6 addresses as well.

It would be a simple matter to save the last successful query time of a device, and then indicate the status of that device on the network maps. The network maps should also indicate which devices are leaf nodes.

Likewise, the interface associated with an ARP entry should be saved. Otherwise, the location of that MAC address may only be known to the VLAN, and not the port itself. Unknown MAC addresses on uplink ports are currently discarded. This is to prevent broken devices from littering our database with bogus data. It is possible for somebody to set up an unmanaged bridge between two managed ones, and if s/he plugs into it, his/her physical location is invisible to GULv3. However, if the information from other switches is available, the MAC address could be saved if it appears on no other leaf port. One would have to

follow the topology to determine which port the MAC address is actually located at.

It would be better if the registration information were obtained more directly than by polling. However, polling is done because of the nature of information available during the time of implementation. There may also be other useful sources of information that could be automatically included in the search. For instance, at USU, there is a database maintained that keeps track of which wall jack a switch port is plugged into. There should be a generic way to include extra sources of information into GULv3.

Currently, whether or not something is considered a searchable term in the display is governed by a set of regular expressions. It would be more useful if a searchable term only showed up as such if it was actually in the database.

Network traffic graphs could be generated per interface, or more likely embedded from another source since there are many solutions that already generate this information [15, 26].

A search on a switch should display the users attached to each port.

CHAPTER 10
CONCLUSION

GULv3, as a project, was a success. It uniquely provides the ability to map both layer-two and layer-three network devices, without relying on STP or on any discovery protocols. In particular, the MAC forwarding table based mapping algorithm described in Section 3.3 is a new way of generating a layer-two topology map without relying on LLDP [4]. It also provides a wealth of network information in a simple format, all using a new type of simple interface. This project ties together many sources of information, and allows the user to organize the information in such a way as to show the relationships among the different searches. This greatly simplifies troubleshooting for network managers and service desk personnel. Indeed, such individuals have come to rely on it in their daily tasks. An objective test was impractical, because there has never been a tool like this on USU's campus with which to compare it to. Even so, the subjective user test shows that users are using it and examining its features, and the access logs show that the usage is increasing. GULv3 will continue to be developed, with user feedback driving most of the features and the bug fixes. Eventually, it will be released as an open source project so that it can benefit anyone who needs it.

REFERENCES

[1] *Hyper-threading technology*, Intel Technology Journal, 6 (2002).

[2] *Ieee standard for local and metropolitan area networks media access control (mac) bridges*, IEEE Std 802.1D-2004 (Revision of IEEE Std 802.1D-1998), (2004). http://ieeexplore.ieee.org/servlet/opac?punumber=9155.

[3] *Ieee standard for local and metropolitan area networks: Overview and architecture*, IEEE Std 802-2001, (2005), pp. 19–20. http://ieeexplore.ieee.org/servlet/opac?punumber=7732.

[4] *Ieee standard for local and metropolitan area networks station and media access control connectivity discovery*, IEEE Std 802.1AB-2005, (2005). http://ieeexplore.ieee.org/servlet/opac?punumber=9971.

[5] *Net-snmp*, Mar. 2007. http://net-snmp.sourceforge.net/.

[6] *Graphviz - graph visualization software*, Mar. 2008. http://www.graphviz.org/.

[7] *Introduction*, Feb. 2008. http://www.cisco.com/en/US/tech /tk648/tk362/tk100/tsd_technology_support_sub-protocol_home.html.

[8] *Mysql: The world's most popular open source databse*, Apr. 2008. http://www.mysql.com.

[9] *Netflow probes: fprobe and fprobe-ulog*, Feb. 2008. http://fprobe.sourceforge.net/.

[10] *Postgresql: The world's most advanced open source database*, Apr. 2008. http://www.postgresql.org.

[11] *pysnmp*, Feb. 2008. http://pysnmp.sourceforge.net/.

[12] I. R. Authority, *Public oui listing*, Feb. 2008. http://standards.ieee.org/regauth/oui/index.shtml.

[13] A. Bierman and K. Jones, *Physical Topology MIB*. RFC 2922 (Informational), Sept. 2000. http://www.ietf.org/rfc/rfc2922.txt.

[14] A. Binczewski, M. Stroinski, and R. Szuman, *Web-based approach to the network physical topology management*, 2002 IEEE Workshop on IP Operations and Management, (2002), pp. 103–107.

[15] T. Bless, *cricket*, Apr. 2003. http://cricket.sourceforge.net/.

[16] E. J. Braude, *The Spiral Process Model*, John Wiley & Sons, Inc., 2001.

[17] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, *Extensible markup language (xml) 1.0 (fourth edition)*, Aug. 2006. http://www.w3.org/TR/2006/REC-xml-20060816/.

[18] J. Case, M. Fedor, M. Schoffstall, and J. Davin, *Simple Network Management Protocol (SNMP)*. RFC 1157 (Historic), May 1990. http://www.ietf.org/rfc/rfc1157.txt.

[19] B. Claise, *Cisco Systems NetFlow Services Export Version 9*. RFC 3954 (Informational), Oct. 2004. http://www.ietf.org/rfc/rfc3954.txt.

[20] Internet Assigned Numbers Authority, *ianaiftype-mib*, Sept. 2007. http://www.iana.org/assignments/ianaiftype-mib.

[21] D. Levi and D. Harrington, *Definitions of Managed Objects for Bridges with Traffic Classes, Multicast Filtering, and Virtual LAN Extensions*. RFC 4363 (Proposed Standard), Jan. 2006. http://www.ietf.org/rfc/rfc4363.txt.

[22] K. McCloghrie and F. Kastenholz, *The Interfaces Group MIB*. RFC 2863 (Draft Standard), June 2000. http://www.ietf.org/rfc/rfc2863.txt.

[23] K. McCloghrie and M. Rose, *Management Information Base for Network Management of TCP/IP-based internets:MIB-II*. RFC 1213 (Standard), Mar. 1991. Updated by RFCs 2011, 2012, 2013.

[24] P. McGuire, *pyparsing*, Sept. 2004. http://pyparsing.wikispaces.com/.

[25] J. Moy, *OSPF Version 2*. RFC 2328 (Standard), Apr. 1998. http://www.ietf.org/rfc/rfc2328.txt.

[26] T. Oetiker, *Tobi oetiker's mrtg - the multi router traffic grapher*, Feb. 2008. http://oss.oetiker.ch/mrtg/.

[27] D. Oran, *OSI IS-IS Intra-domain Routing Protocol*. RFC 1142 (Informational), Feb. 1990. http://www.ietf.org/rfc/rfc1142.txt.

[28] P. Phaal, S. Panchen, and N. McKee, *InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks*. RFC 3176 (Informational), Sept. 2001. http://www.ietf.org/rfc/rfc3176.txt.

[29] R. Presuhn, *Management Information Base (MIB) for the Simple Network Management Protocol (SNMP)*. RFC 3418 (Standard), Dec. 2002. http://www.ietf.org/rfc/rfc3418.txt.

[30] D. Raggett, A. L. Hors, and I. Jacobs, *Html 4.01 specification*, Dec. 1999. http://www.w3.org/TR/html401/.

[31] J. Raskin, *The Humane Interface*, Addison-Wesley Professional, Apr. 2000.

[32] G. Roelofs, *Portable network graphics: An open, extensible image format with lossless compression*, Feb. 2008. http://www.libpng.org/pub/png/.

[33] S. Routhier, *Management Information Base for the Internet Protocol (IP)*. RFC 4293 (Proposed Standard), Apr. 2006. http://www.ietf.org/rfc/rfc4293.txt.

[34] P. Srisuresh and K. Egevang, *Traditional IP Network Address Translator (Traditional NAT)*. RFC 3022 (Informational), Jan. 2001. http://www.ietf.org/rfc/rfc3022.txt.

[35] A. S. Tanenbaum, *The Network Layer in the Internet*, Prentice Hall PTR, 2003.

[36] J. Tidwell, *Designing Interfaces*, O'Reilly Media, Nov. 2005.

[37] Van Jacobson, *traceroute*, Feb. 1989. ftp:/ftp.ee.lbl.gov/traceroute.tar.gz.

[38] R. Wagner and J. Bryner, *Address resolution protocol spoofing and main-in-the-middle attacks*, Information Security Reading Room, (2006). http://www.sans.org/reading_room/whitepapers/threats/474.php.