2015-06-01

Service Dependency Analysis via TCP/UDP Port Tracing

John K. Clawson
Brigham Young University - Provo

Service Dependency Analysis via TCP/UDP Port Tracing

John K. Clawson

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Joseph J. Ekstrom, Chair
Derek L. Hansen
Kevin B. Tew

School of Technology

Brigham Young University

June 2015

ABSTRACT

Service Dependency Analysis via TCP/UDP Port Tracing

John K. Clawson
School of Technology, BYU
Master of Science

Enterprise networks are traditionally mapped via layers two or three, providing a view of what devices are connected to different parts of the network infrastructure. A method was developed to map connections at layer four, providing a view of interconnected systems and services instead of network infrastructure. This data was graphed and displayed in a web application. The information proved beneficial in identifying connections between systems or imbalanced clusters when troubleshooting problems with enterprise applications.

Keywords: servicemap, layer 4, dependency discovery, TCP, UDP, topology map, tracing

ACKNOWLEDGEMENTS

Thanks to John Payne, director of Infrastructure Engineering at BYU’s Office of IT, as well as the other members of the Platform Linux team for their suggestions, input, and testing of this tool.

A special thanks to Dr. J.J. Ekstrom for continued guidance and insight to refine and guide the research along with the other committee members.

Lastly, many thanks to my dear wife Rachel, who gave up many hours of my time so that this could happen and put in hours of her own outstanding proofreading work.

TABLE OF CONTENTS

ABSTRACT ii

ACKNOWLEDGEMENTS iii

TABLE OF CONTENTS iv

LIST OF FIGURES viii

LIST OF TABLES ix

1 Introduction 1

1.1 Background 1

1.2 Problem Statement 3

1.3 Hypotheses 4

1.4 Definitions 4

1.5 Justification 5

1.6 Summary of Proposed Methodology 5

1.7 Scope 6

1.8 Delimitations 6

1.8.1 Automated Parsing of Generated Information 6

1.8.2 Generation of Additional Data 6

1.8.3 Agentless Data Collection 7

1.8.4 Agents for Non-Linux Operating Systems 7

1.8.5 Performance Monitoring 7

1.8.6 System Discovery 7

2 Literature Review 8

2.1 OSI Model 8

2.2 Layer 2 Mapping 10

2.3 Layer 3 Mapping 10

2.4 Layer 4 Mapping 11

2.5 Layer 7 Mapping 14

2.6 Current Options 14

2.7 Literature Review Conclusions 15

3 Methodology 16

3.1 Improving Upon Past Ideas 16

3.2 Software Functions 16

3.2.1 Host Agent 17

3.2.2 Collector 17

3.3 Performance Requirements 17

3.3.1 Agent Performance 17

3.3.2 Collector 18

3.4 Test Environment 18

3.5 Analysis Framework 18

3.5.1 Performance 18

3.5.2 Accuracy 19

3.5.3 Utility 19

3.6 Costs vs Benefits 20

4 Implementation 21

4.1 Architecture Overview 21

4.2 Agent 21

4.2.1 Iptables Configuration 22

4.2.2 RSYSLOG Configuration 22

4.3 Collector.. 23

4.3.1 RSYSLOG Configuration 23

4.3.2 Database Configuration 23

4.3.3 Hourly Summary Job 24

4.3.4 Web Application 25

5 Results and Data Analysis 29

5.1 Functionality 29

5.2 Performance 30

5.2.1 CPU and Memory Utilization 30

5.2.2 Network Bandwidth 31

5.2.3 Database Storage Requirements 31

5.3 Validity 31

5.3.1 Manual Connection Discovery Comparison 32

5.3.2 Alternative Tool Comparison 33

5.4 Utility 34

5.4.1 Survey Results 35

5.5 Costs Versus Benefits 36

6 Conclusions and Recommendations 37

6.1 Conclusions 37

6.2 Future Work 37

References 41

Appendices **43**

Appendix A. Source Code for Linux Agent **44**

Iptables Configuration 44

RSYSLOG Rule File 44

Appendix B. Source Code for Collector **45**

RSYSLOG Configuration 45

Database Create Scripts 45

Hourly Summary Job 45

Web Application HTML 48

Web Application CGI Host List 54

Web Application CGI Port List 56

Web Application CGI Connection List 57

LIST OF FIGURES

Figure 2-1: OSI Model 9

Figure 4-1: Architecture Overview 21

Figure 4-2: Web Application Interface 27

Figure 5-1: Sample Graph 29

LIST OF TABLES

Table 1: LogSummary Table Structure 24

Table 2: System Specification and Utilization 30

Table 3: Manual Accuracy Verification 33

Table 4: Product Accuracy and Cost 34

1 INTRODUCTION

1.1 Background

There is a wealth of information about enterprise IT systems that is not currently easily exposed. An example is information about service dependencies between hosts. Gathering that knowledge and assembling it into a useful form has many benefits including reduced errors and better security.

The popularity of virtualization, cloud models, and web services has resulted in the number of systems increasing many times over. In BYU's Office of IT, the number of Linux servers has grown from one to two hundred in 2008 to more than a thousand Linux servers in 2014. This scale makes troubleshooting by hand more tedious and error-prone than ever. Automated tools to gather and report the necessary data are the only long-term viable solution to maintaining large numbers of systems without a corresponding increase in the number of administrators. In that time frame the number of Linux systems engineers has increased to handle the additional duties and workload, but the requirement is now to do more without increased human capital.

Change management, particularly validating that a proposed change won't have any unplanned or undesired side effects, is the problem this research seeks to solve. When a system administrator needs to change a firewall or move a host to another subnet, he needs to understand all host communications impacted by the change to make sure nothing is configured incorrectly. Experience shows that this work is generally performed from memory, meaning the administrator relies only on his familiarity with and previous knowledge of the host. That knowledge may be inaccurate or outdated or simply forgotten. Many errors will be avoided if the system administrator uses a more reliable way of validating that his knowledge of host connections is true, or discovering that there are unaccounted-for gaps in his knowledge before making critical changes.

A connection is defined for the purpose of this research as a network flow identified by the layer 3 (IP address) and layer 4 (port and protocol) components.

Much of the information that will aid system administrators in making accurate changes can be found by examining the connections on the OSI layer 4. Determining network dependencies and communications between hosts on layer 4 (generally over TCP and UDP protocols) is already a common practice that many system administrators do by hand when they are troubleshooting problems. The problem with determining dependencies manually is that the administrator must choose between a fast, potentially inaccurate result and a slow and laborious but accurate result. A fast but inaccurate example would be running the netstat command. It will capture all connections in use, but that is only one snapshot in time and will fail to accurately represent all of the services provided by or consumed by a system. A slow example would be running a packet capture with a tool like tcpdump, then parsing the captured data with other tools like Wireshark or custom scripts to assemble a list of connections made over a period of time. Those methods are also limited to a specific time range, albeit one defined by the user.

A number of tasks depend on this knowledge of service dependencies such as firewall and security configurations and outage resolution. One scenario is when the system administrator needs to validate whether a particular service is being used, or which hosts are consuming it before he may safely remove a firewall rule that allowed access to it. Many connections are short-lived and will be missed by spot checks, so a quick manual check is not likely to reveal all of the services that a host regularly consumes. Change management is the driving issue behind this research, but there are many potential benefits and uses for such knowledge.

There are products to map other layers of the OSI model, but none that fill the need of enumerating the service dependencies (layer 4) between hosts. Apart from the OSI layer 4, there have been many approaches to mapping networks using data from layers 2 (physical network) and 3 (virtual network) that have become mature products like Spiceworks. However, the existing products have a gap in functionality: they are generally targeted at network device management and inventory, not service dependency analysis. They do not map the logical network on layer 4. On the other end of the OSI model are products to map application-specific data flows (layer 7), and while many applications provide data about their own data flows and connections, they cannot see outside of their application containers to provide a comprehensive view that will assist an system administrator in ensuring changes are accurate. There are few products currently available that gather data about layer 4 connections, and those that do gather the data use it for inventory management and don't expose it in an easy format to assist system administrators.

In short, the system administrator must either manually connect to a host to spot-check connections manually or rely on memory. Both of those approaches are error prone, leading to a potential combination of accidental service outages and potentially reduced security posture as changes are made to systems and hosts. The solution proposed in this paper, if used by system administrators, will reduce that potential for error by automating that process and making the results more accessible.

1.2 Problem Statement

Developers and system administrators do not have a comprehensive view of layer 4 connections between servers and clients. This lack is an obstacle to best development practices, security, and troubleshooting. In order to make accurate system configurations and commit fewer errors a tool is needed that provides a concise and accurate view of the connections that are actively used between hosts.

1.3 Hypotheses

Hypothesis 1 (H1): It is possible to create an extensible model to describe dependency relationships between computer systems.

Hypothesis 2 (H2): Systems can do some level of automatic dependency/interaction reporting on layer 4 to populate relationship models programmatically.

Hypothesis 3 (H3): Topology maps assembled from system-reported data will give a useful and accurate picture of the environment.

Objective 1 (O1): Build a method of reporting connections to a central collector.

Objective 2 (O2): Build a collector to store the data.

Objective 3 (O3): Build a web application to provide useful views into the topology.

1.4 Definitions

Topology Map -- A visual representation of how systems and servers are interconnected.

Service -- A network service that may be consumed by either the local host or another network host (i.e., the process that receives connections on a listening port).

Client -- The dependent host or process that is consuming a network service (i.e., the process that initiates connections from an ephemeral port).

Layer 4 -- The OSI model layer that is largely comprised of TCP and UDP connections.

Host Agent -- The software component that runs on a server to collect data about layer 4 connections. Also referred to as “agent”.

Collector -- The software component that gathers data from host agents, stores, and correlate the data.

1.5 Justification

There is a gap in easily accessible information about how computer systems in an enterprise network interact. While there are many products to map the network on layer 2 and layer 3, there are not many options for doing so on layer 4. Assembling a comprehensive view of layer 4 communications that will improve change management by providing more information to system administrators before a configuration is changed. This will prevent errors and reduce downtime.

1.6 Summary of Proposed Methodology

Create a conceptual model that allows for dependency relationships. The model will be a simple description that establishes the working vocabulary and allows the data to be arranged in a way that helps users understand the systems.

Write proof-of-concept tool to collect connection data and create dependency maps. An example implementation is essential to prove that the model and method of dependency discovery will actually work in the real world. The tool will be developed in an iterative cycle with feedback collected from OIT system administrators. A team of interested parties has been put together to analyze the tool and make suggestions about its development. Feedback from monthly progress meetings will be documented along with the response to it, be it resolution or a determination that it is invalid.

Verify accuracy by comparing results against manually-gathered data. A randomly-selected set of 10 systems will be checked by hand to verify that the automated reporting is working properly.

Talk with users of the tool to verify whether or not it is easy to use and provides helpful information. This will be done by presenting the tool to 5 system administrators and gathering their feedback in a short survey. I will also evaluate the tool myself as a user and provide feedback.

1.7 **Scope**

This research will be limited to the discovery and mapping of connections at the layer 4 (TCP/UDP) level and displaying those connections for system administrators to use.

1.8 **Delimitations**

The following are outside of the scope of this research.

1.8.1 Automated Parsing of Generated Information

A desired outcome of this research is to produce a useful graphical representation of the layer 4 connection map. Further use of the data is beyond the scope of this research.

1.8.2 Generation of Additional Data

There are many existing products to generate and parse data about connections at layers 3 and 2. That topic is well developed, and will not be replicated. Application-specific instrumentation also exists to collect data at layer 7. There are also other protocols that may potentially be used at the layer 4 level besides TCP and UDP. However, TCP and UDP comprise the vast majority of Internet and intranet traffic, and others will be excluded from this research.

1.8.3 Agentless Data Collection

An extension of this research would be to develop a system of collecting data from systems that do not support an agent (or special configuration on the system). This generally entails a remote login from the collector to gather the data.

1.8.4 Agents for Non-Linux Operating Systems

An agent will be written to work on a version of Linux. Agents for the Windows or other operating systems are out of scope.

1.8.5 Performance Monitoring

Collecting data to provide performance metrics and reporting is beyond the scope of this research. This would be a topic for further research, as it is one of the next logical steps once the basic topology is known. Performance monitoring could be combined with an automated root cause analysis engine to localize problems and bottlenecks.

1.8.6 System Discovery

A number of products on the market that are aimed at layer 2 and layer 3 mapping incorporate scanners to find systems or devices on the network that the operator may not have known about (network inventory). This research does not deal with that inventory, but on gathering new data from known systems.

2 LITERATURE REVIEW

What follows is a review of the development of network models, mapping technologies, and available products.

2.1 **OSI Model**

The Open Systems Interconnection (OSI) subcommittee was created by the International Organization for Standardization (ISO) in 1977. Their first priority was to create an architecture that would provide a framework upon which to implement standard protocols (Zimmermann 1980).

The following image from Wikimedia commons illustrates the OSI model. The layers are counted from bottom to top so that layer 1 is on the bottom. Layer 4, the focus of this research, is the “transport” layer in Figure 2-1. It is the lowest level layer that resides on the host instead of on the network.

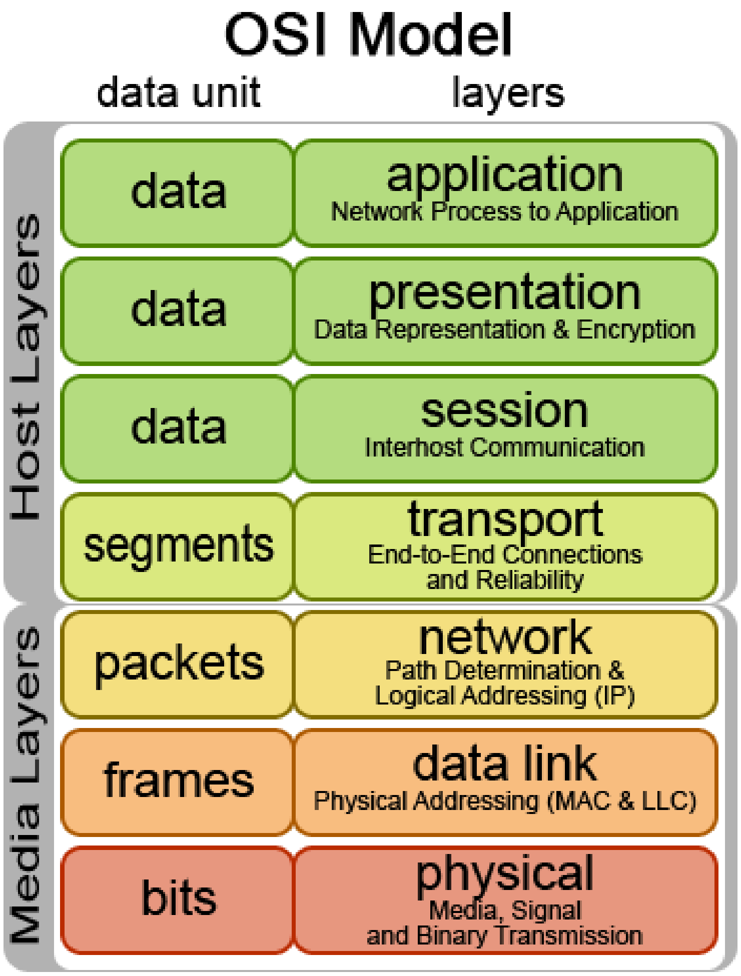

OSI Model
data unit
layers
Host Layers
Media Layers
data
application
Network Process to Application
data
presentation
Data Representation & Encryption
data
session
Interhost Communication
segments
transport
End-to-End Connections and Reliability
packets
network
Path Determination & Logical Addressing (IP)
frames
data link
Physical Addressing (MAC & LLC)
bits
physical
Media, Signal and Binary Transmission

Figure 2-1: OSI Model

The fact that layer 4 is the lowest layer on host devices and not on network devices provides incentive to utilize a host-based agent framework for reporting, as the host is where the most accurate data will be found.

Layers 2 and 3 are commonly mapped, as the technology to do so is well developed. Layers 4-7 (everything on the host side) are where things get complicated. For simplicity, most implementations of the network stack actually use the TCP/IP model that groups layers 5-7 of the OSI model together. However, the OSI model is the reference framework, so it will be used as the foundation when referencing and discussing these topics.

2.2 Layer 2 Mapping

Layer 2 network mapping is done by using the protocols that the switches use to configure themselves, such as CDP, JDP, and LLDP (Spiceworks 2014). The layer 2 map shows the physical layout of the network by describing what devices are connected to which ports on a switch or router.

2.3 Layer 3 Mapping

Layer 3 network mapping is a necessary feature for any network administrator. It shows the logical segmentation of the network and describes reachability between hosts and network segments. This is generally accomplished by using SNMP queries to network switches and routers, active probes to endpoints, and route analytics (Spiceworks 2014).

Software packages that do network-oriented mapping usually include both layer 2 and layer 3 and tie the information together to aid performance analysis and root cause detection. This is good information, but does not include data about how services are consumed.

Examples of this type of software suites include:

- HP OpenView (Hewlett-Packard 2015)
- Lumeta IPsonar (Lumeta 2015)
- NetCrunch (AdRem 2015)
- Nmap (Nmap 2015)
- PacketTrap (PacketTrap 2015)
- Scrutinizer (Scrutinizer 2015)
- SolarWinds (Solarwinds 2014)
- Spiceworks (Spiceworks 2014)

- WhatsUp Gold (WhatsUp Gold 2014)
- Docusnap (Docusnap 2015)

2.4 Layer 4 Mapping

As the number of systems in use has grown over the years with the explosion of the Internet, so has interest in mapping interactions at layer 4. There have been a number of academic research projects and papers in the last decade that have addressed the problem in some form or another. Some of those, such as the Orion project developed by the University of Michigan in conjunction with Microsoft Research, have been integrated into commercial products (Orion is part of SolarWinds now).

Because these past projects are foundational to this thesis, they will be discussed individually.

Sherlock is a system developed by Microsoft Research that uses a host-based agent to monitor network traffic and correlate interactions between hosts as dependencies. Sherlock is a complex system designed to report not just failures, but also performance degradations. To determine dependency data, Sherlock monitors the network packets sent and received by each host to figure out what dependencies that host has (Bahl et al. 2007). Unfortunately, using a host agent to capture full network data simply adds complication and maintenance since the same data could be monitored on a network capture device. Sherlock attempts to correlate dependencies via layer 3 interactions instead of layer 4.

Constellation is another system developed by Microsoft Research that aims to identify dependencies among network hosts with an aim of learning what systems depend on. Its function is purely to map the network from a service/functional perspective. It treats the network as a black box and builds its dependency map solely from network data captured on a switch or router.

Constellation uses the timing of packet transmission and reception to determine dependencies. Because of this, however, it is making a “best guess” effort and may not be accurate, potentially requiring more human intervention and effort to be useful (Barham and Black 2008).

A system called Orion was developed between the University of Michigan and Microsoft Research whose goal is to create a dependency map. Orion is similar to Constellation, and watches network traffic for layer 3/4 headers to determine dependencies. The authors considered an option of mining application configuration files, but determined that this would be too limiting and would not work well to determine dynamic or runtime dependencies. Specifically, the authors settled on watching IP, TCP, and UDP headers combined with timing data in order to construct the model they wanted.

The authors of the Orion system made a few important notes. It is difficult to infer dependencies from application traffic (layer 7) unless intelligence about a particular application has been codified. This means that creating a system that understands all of the applications it monitors requires a significant amount of manual effort. They also note that timing data or packet headers alone, which have been frequently used to correlate messages in other systems of this type, are subject to variance due to load and other factors in the network that make them unreliable. That unreliability causes either false positives or missed positives, depending on the tuning (Chen et al. 2008).

Orion does produce what is likely a viable dependency map, and indeed was good enough to be incorporated into a commercial product. However, the same result can most likely be accomplished by tackling the problem from a different perspective and leaving timing out of the equation entirely.

A paper by Tobias Binz discusses a plugin-based system that pulls information from host agents to assemble what he calls an enterprise topology graph (ETG). By having a plugin-based system with an open specification, users are able to write and test their own plugins for the system to extend it to meet their needs (Binz et al. 2013). The framework and plugins they created is possibly a bit too software/service-specific and not generic enough. This comes closest to hitting the mark for being flexible and extensible. However, it requires a lot of additional development in the way of plugins to add data about all of the particular applications that may exist in a given enterprise network. While the Orion authors expected that the approach used by Binz would not work due to the required manual effort, it’s possible that with a solid open-source community behind it to support plugin development it might reach viability. Otherwise, it is a good technical solution but not likely capable of succeeding outside of the lab.

ServiceNow Discovery purports to do application-interaction discovery with an agentless system. It requires credentials for remote logins to run commands such as netstat to gather the data. However, this may miss short-lived connections and ServiceNow does not support Linux targets (ServiceNow 2014).

VMware’s vCenter Application Discovery Manager software worked by capturing network data from VMware’s virtual network devices (VMware 2014). The white papers do not go into detail about how it analyzes the data behind the scenes, but it can be presumed to have the same pitfalls as other network capture systems. This product was also discontinued as of 1 Jun 2013 with no planned functional replacement.

A product called JDisc Dependency Mapping for JDisc Discovery Inventory Solution says it can query connection tables via SNMP to gather the data it needs (JDisc 2014). This may also prove to miss short-lived connections.

There are a few other commercial products that claim to have the desired capabilities, but a review of their features and how they work reveals that they have chosen compromises similar to a few of the academic projects already discussed.

2.5 **Layer 7 Mapping**

Layer 7 of the OSI model is the application layer. There are a number of products capable of instrumenting and mapping the data flow of a network application. The most common example is web service mapping. A 2008 IEEE conference paper by Sujoy Basu discussed options for dynamically determining dependencies among web services by observing the requests at the application layer instead of the network layer (Basu, Casati, and Daniel 2008).

The approach used by these and other middleware/application layer mapping products is usually to inject code into the application itself that reports on connections and status. While this is certainly the most accurate and reliable method, it is not feasible to apply to the wide variety of products found in a typical enterprise network. The variety of applications that run in an enterprise environment, combined with the fact that many of them are not written in-house makes it impossible to attempt to instrument them all at the code level.

In the case of a fairly homogenous environment where such instrumentation may be feasible, there are products such as AppNeta's TraceView and AppView solutions that utilize new code to inject "dye" into the network that allows for performance and topology monitoring (AppNeta 2014).

2.6 **Current Options**

The only current commercial option for customers who want to gather and view data about layer 4 service dependencies between hosts is SolarWinds Orion. However, its mapping tools are still focused on geographic and physical layouts, making it ill suited for a service-level perspective. Due to the way it stores full network streams, it also requires expensive hardware to deploy.

For those who are willing to put in a bit more work and be satisfied with a data-only view, it's possible to use traditional security information and event management (SIEM) products to gather full firewall logs and then write custom filters to generate alerts from those. However, those products, such as Tenable's Log Correlation Engine (Tenable 2014) and SolarWinds' Log & Event Manager (Solarwinds 2014) are built with different goals in mind and would not provide useful graphical views. This is a situation that illustrates the differences between gathering data (many SIEM products are capable of gathering the necessary data after custom configuration) and providing useful views into information.

2.7 Literature Review Conclusions

The problem space of discovering layer 4 dependencies is not new, and there have been a number of attempts to solve the problem both in the academic and commercial circles. Orion is the only programmatic solution that has been commercialized, and it was folded into a network performance management suite that does not advertise those capabilities to system administrators. The site for that product does not even reference this capability directly. Other commercial products involve a prohibitive amount of manual configuration. Several of the academic research projects looked promising, but all have tradeoffs. Some do not produce results that are accurate enough to be trusted, while others require too much manual intervention.

There is still a lot of space for new research in this field. What has been done so far is really just the beginning, and this field will be forced to mature quickly along with the effort to migrate systems to cloud infrastructures. As systems of systems become more complex an automated way of assembling layer 4 dependency information will be necessary.

3 METHODOLOGY

The required software functions, performance, and use cases are described. A necessary test environment is planned.

3.1 Improving Upon Past Ideas

All of the previous attempts to solve this problem have a series of tradeoffs. I do not expect anything produced at the current time to be an exception to that. However, I do believe that there are sets of tradeoffs that have not yet been explored, and may produce a more desirable set of outcomes for important and common use cases.

One of the problems with a number of proposed solutions is that they try to do too much. They attempt to be the one-stop product for monitoring networks, including dependency mapping and performance monitoring.

By choosing to focus solely on dependency mapping between systems and ignoring other aspects such as performance data and integration with other layers of mapping and monitoring suites at first, I expect to produce something more immediately useful. This will be a depth-first approach instead of a breadth-first approach.

The model and framework for storing the collected data will be designed to allow for flexibility and the addition of future data types once the core product is finished.

3.2 Software Functions

There are two types of software that will be written: host agent and collector.

3.2.1 Host Agent

The agent will monitor TCP and UDP connections on the host and send the data to the collector. Specifically, the host agent will consider listening ports as services and host-initiated connections as clients. The host agent will have a mechanism to determine if a connection on an ephemeral port is part of a service (i.e., the connection is part of a request that was initially received on the listening port) or if it is part of a client request to another host.

This information will be sorted into services and clients, so that the agent reports to the collector the services that the host provides and the other hosts it depends on as a client.

3.2.2 Collector

The collector will receive data from the host agents and apply logic to build useful information from it.

A data storage component will provide a repository for the data received from host agents and facilitate backup and export functions.

A web application component will provide a graphical view into the collected data and have the ability to drill down into the network and show different views. Examples may be the option to see dependencies per host or to see all of the dependencies behind an endpoint.

3.3 Performance Requirements

3.3.1 Agent Performance

The host agent or other method of querying the target machine must not consume more than 10% of the compute resources on a reference system, even during peak usage.

3.3.2 Collector

The collector's only performance requirement is that it be able to keep up with the data sent from a large set of machines. One collector on a reference system (two virtual CPUs) must be sufficient to receive data from at least 1000 agents.

It is feasible that during a large burst of network activity data might be lost, but even in that case it will have little to no effect on the resulting information. The tool is designed to show layer 4 interactions, not forensic data. With service dependency connections repeating regularly, occasional and brief data loss is not concerning. Over time any occasional failures will be insignificant.

3.4 Test Environment

The test environment will be comprised of at least twenty non-production virtual Linux servers in an enterprise environment.

3.5 Analysis Framework

The developed tool will be judged on three broad categories: performance, accuracy, and utility.

3.5.1 Performance

Performance will be based on an arbitrary limit such that a typical system might run the software without interfering with its main operations. The limit will be determined by discussing the topic with a team of senior system administrators.

The host agent must be able to run in the background without consuming more than 5% of the total CPU cycles on a dual-core reference platform.

The collector must be capable of handling a large number of client systems (500+) without exceeding the resources available on the dual-core reference platform.

3.5.2 Accuracy

Accuracy will be determined by comparing the results of manual layer 4 connection discovery with the results reported by the tool. A random set of twenty systems will be selected and a list of service dependencies manually generated. The manual results will be compared to the automated ones and any missing data from either set calculated.

The accuracy of the developed system will be compared to the alternatives that were discussed in the literature review. The new method will be considered successful if the accuracy is higher than that of other systems, or if it is equal but has other benefits such as ease of use and lower maintenance.

There may be a tradeoff between resource consumption and accuracy. If the host agent is constantly running on a busy system it may result in undesirable overhead. In that case an option could be built to provide a slower checking mechanism, but one that will still prove accurate when run for a sufficient sample period on the host system.

3.5.3 Utility

The users of the tool (system administrators) will be given an evaluation period and asked to use the tool as part of their change management routine. They will be asked to complete a short survey about the utility of the tool and any other feedback they may have.

3.6 **Costs vs Benefits**

There are no special hardware requirements for a tool of this nature. However, it is important to make it as accessible and easy to deploy and maintain as possible to ensure that users aren’t forced to make a difficult choice between the effort to deploy or maintain the tool and the benefit that is gained from it.

4 IMPLEMENTATION

4.1 Architecture Overview

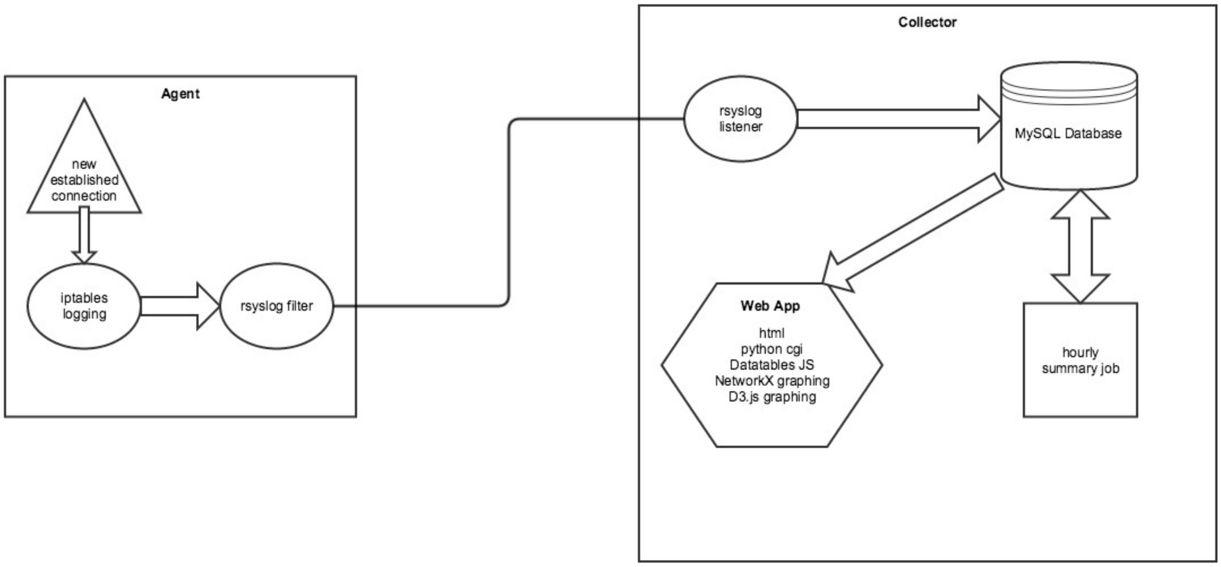


Figure 4-1: Architecture Overview

4.2 Agent

After considering a number of options ranging from instrumenting the kernel to writing custom software, the best approach found was to use built-in utilities. Both Linux and Windows support native firewall logging. The firewall is a logical choice for monitoring connections since it already tracks all incoming and outgoing connections. The Linux agent was developed for demonstration purposes.

The Linux agent is a combination of iptables firewall log rules and RSYSLOG filtering to gather the data and send only relevant results to the collector.

4.2.1 Iptables Configuration

A challenge encountered while developing the host agent was the need to correctly log which ports were the source and destination in a stream. Many hours of research into iptables mechanisms were spent in an effort to log only inbound connections to listening ports or outbound connections to listening ports, excluding the ephemeral/client side of the conversation. Unfortunately, this proved extremely difficult and unreliable due to the nature of iptables and netfilter connection tracking. The final solution was to log everything regardless of client/server status, then sort it out in the hourly summary job. This proved much more reliable and far less invasive to implement. A downside was the potential to require manual corrections for some connections. For example, NFS connections utilize the portmapper daemon, so while the server side of the connection is always on port 2049, the client side is initiated on a high numbered ephemeral port but then switches to a port between 600-1024. This required a manual tweak in the summary job to correctly represent the connections. The method used for the iptables rules is only valid on iptables 1.4 or newer.

In order to separate the logs generated for this purpose from all others, a unique tag string was added to the beginning of the log message. The string, "newConn", allowed RSYSLOG to match just those logs and parse them accordingly. In order to log each established connection only once and reduce overall traffic, the iptables connection mark is used to mark connections that have been logged previously. All packets are then checked for the mark before logging, and if the packet has the mark it is not logged. The source code for the Linux agent is included in Appendix A.

4.2.2 RSYSLOG Configuration

RSYSLOG is a logging system that is the default on most modern Linux distributions. The iptables rules logged to the kernel.info target, which was parsed by RSYSLOG. A rule file was written for RSYSLOG that when the “newConn” tag was found in a message, that message would be forwarded to the collector and then discarded. The messages were not stored locally on the agents in order to prevent filling local storage volumes.

4.3 **Collector**

The collector consisted of an RSYSLOG daemon listening for UDP traffic and a MySQL database to store the messages as they were received. An hourly summary job ran to parse the new messages in the database. A web application provided a graphical and table view into the resulting data. The source code for the collector is included in Appendix B.

4.3.1 RSYSLOG Configuration

RSYSLOG was configured to find strings matching the “newConn” tag applied by iptables. Messages matching that string were forwarded over UDP to the collector and then discarded to prevent them being logged on the local filesystem. The UDP transport was chosen over TCP as it has much lower network overhead and the guaranteed reception of TCP was not necessary. If messages occasionally did not arrive at the collector, the overall set of data and accuracy of the set will be unaffected, as the type of data being collected was repeated.

4.3.2 Database Configuration

The database chosen was MySQL. It was configured with the database create script provided by RSYSLOG, which creates a Syslog database and SystemEvents table that RSYSLOG where records information.

A LogSummary table was added to the Syslog database to store the processed records. The table structure follows.

Table 1: LogSummary Table Structure

Field	Type	Comment
id	int	unique ID for database reference
scrip	varchar	Source IP address
dstip	varchar	Destination IP address
proto	varchar	Layer 4 protocol
srcport	int	Source (ephemeral) port
dstport	int	Destination port
counter	int	How many times a connection has been recorded
lastseen	datetime	Timestamp of the last recorded a connection
firstseen	datetime	Timestamp from the first time a connection was recorded

A unique connection (network flow) was defined by the combination of the source IP address, destination IP address, protocol, and destination port. This is the model that may be exended with additional data if required for integration with future data. The other fields were recorded for informational purposes.

4.3.3 Hourly Summary Job

The Python program connectionSummary.py was run every hour by a task scheduler (cron). The source code is included in Appendix B. The program parsed the data collected in the default Syslog database into the LogSummary table. This was done to reduce the volume of the storage required, as the iptables logs would otherwise consume several gigabytes per day with a small set of just 40 agent systems. This program depended on the mysql-connector-python package being installed.

The log format, like most *nix system logs, was done in plain text. The RSYSLOG database contained a table called SystemEvents that holds all of the received events, with the Message field containing the text string received from the remote system. To parse this, the summary program split the text string into components and looked for text matches to assign to fields in the summary table LogSummary. The message logged by iptables contained the network interface name, whether the packet was inbound or outbound, the source and destination ports and IP addresses, the protocol, and a few other packet-related details.

A unique connection was identified by the set of destination IP address, source IP address, protocol, and destination port. If a connection was a repeat of one that had been summarized previously, then the count field was incremented and the last-seen timestamp was changed to the one on the latest message. If it was a new connection that had not been seen before, a new row was created in the summary table with all of that information as well as the first time the connection was seen.

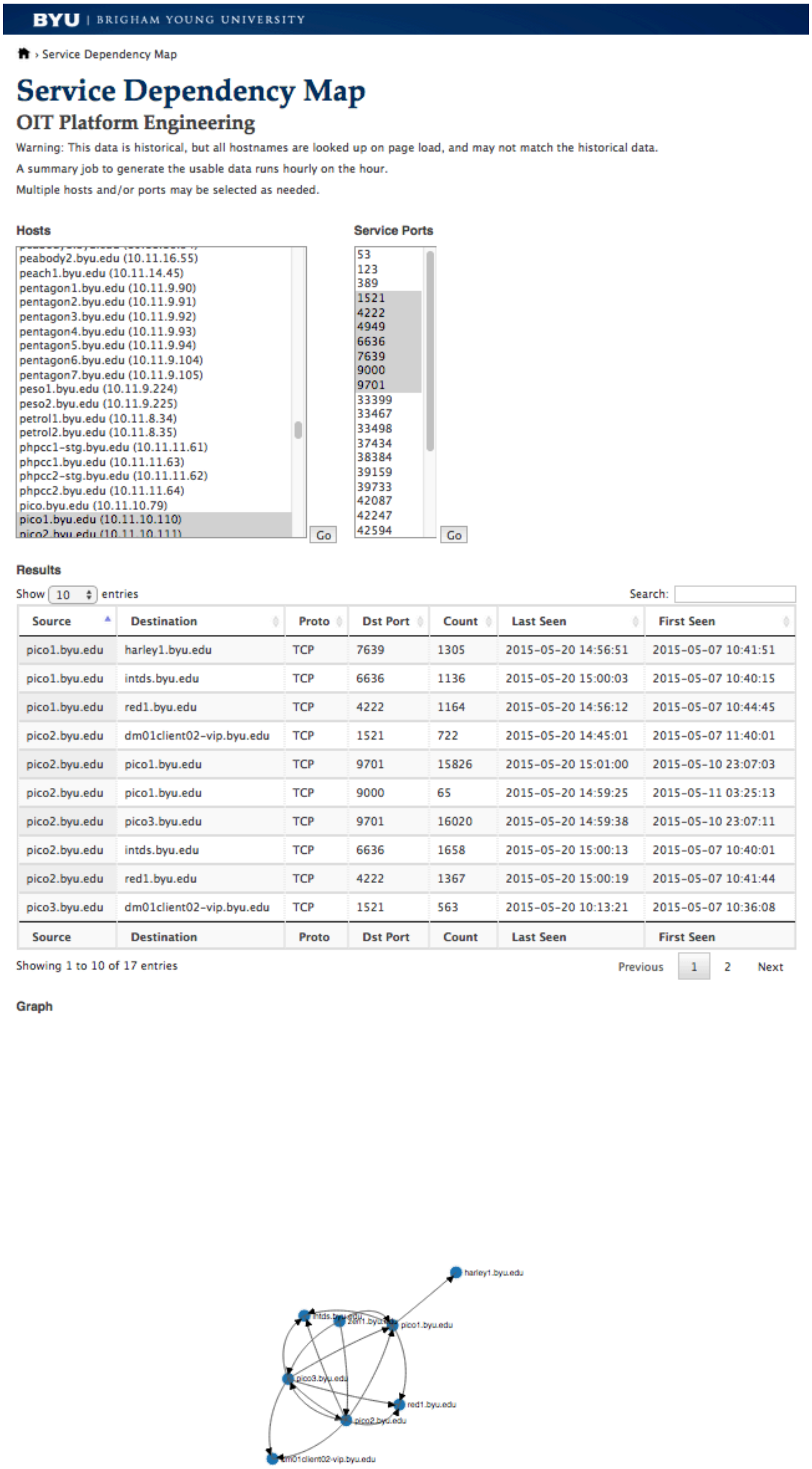
4.3.4 Web Application

The web application consisted of HTML, JavaScript, and Python CGI scripts running on an Apache web server. An HTML page provided the visual interface and made AJAX calls to a collection of Python CGI scripts to retrieve formatted data from the database.

The web application went through a number of revisions. The first attempt was to just create a visual graph of any connection related to a hostname that was typed into a text field. That proved to be problematic, as a single host can easily have hundreds of connections. The visual graph is an excellent way to convey data that is difficult to understand from numbers in a table, but it is only effective if there are fewer than a dozen or so nodes and edges on the graph. With even a single node selected that had many dependencies, the graph was unreadable.

The initial version was therefore scrapped and a new web application was written that allowed the user to select from a list of all hosts in the database the particular ones they were interested in, and then select a list of service ports that they wanted to see more information about.

Once they did that, a table was populated with the results and a graph was drawn that showed only the results of their filtered selections. This provided the best of both worlds by allowing them to see data in the graph just a few nodes and services were selected, and to see numerical data displayed in the table that can also be filtered and sorted (see Figure 4-2).



BYU | BRIGHAM YOUNG UNIVERSITY

› Service Dependency Map

Service Dependency Map

OIT Platform Engineering

Warning: This data is historical, but all hostnames are looked up on page load, and may not match the historical data.

A summary job to generate the usable data runs hourly on the hour.

Multiple hosts and/or ports may be selected as needed.

Hosts

peabody2.byu.edu (10.11.16.55)
peach1.byu.edu (10.11.14.45)
pentagon1.byu.edu (10.11.9.90)
pentagon2.byu.edu (10.11.9.91)
pentagon3.byu.edu (10.11.9.92)
pentagon4.byu.edu (10.11.9.93)
pentagon5.byu.edu (10.11.9.94)
pentagon6.byu.edu (10.11.9.104)
pentagon7.byu.edu (10.11.9.105)
peso1.byu.edu (10.11.9.224)
peso2.byu.edu (10.11.9.225)
petrol1.byu.edu (10.11.8.34)
petrol2.byu.edu (10.11.8.35)
phpcc1-stg.byu.edu (10.11.11.61)
phpcc1.byu.edu (10.11.11.63)
phpcc2-stg.byu.edu (10.11.11.62)
phpcc2.byu.edu (10.11.11.64)
pico.byu.edu (10.11.10.79)
pico1.byu.edu (10.11.10.110)
pico2.byu.edu (10.11.10.111)

Go

Service Ports

53
123
389
1521
4222
4949
6636
7639
9000
9701
33399
33467
33498
37434
38384
39159
39733
42087
42247
42594

Go

Results

Show 10 entries

Search:

Source	Destination	Proto	Dst Port	Count	Last Seen	First Seen
pico1.byu.edu	harley1.byu.edu	TCP	7639	1305	2015-05-20 14:56:51	2015-05-07 10:41:51
pico1.byu.edu	intds.byu.edu	TCP	6636	1136	2015-05-20 15:00:03	2015-05-07 10:40:15
pico1.byu.edu	red1.byu.edu	TCP	4222	1164	2015-05-20 14:56:12	2015-05-07 10:44:45
pico2.byu.edu	dm01client02-vip.byu.edu	TCP	1521	722	2015-05-20 14:45:01	2015-05-07 11:40:01
pico2.byu.edu	pico1.byu.edu	TCP	9701	15826	2015-05-20 15:01:00	2015-05-10 23:07:03
pico2.byu.edu	pico1.byu.edu	TCP	9000	65	2015-05-20 14:59:25	2015-05-11 03:25:13
pico2.byu.edu	pico3.byu.edu	TCP	9701	16020	2015-05-20 14:59:38	2015-05-10 23:07:11
pico2.byu.edu	intds.byu.edu	TCP	6636	1658	2015-05-20 15:00:13	2015-05-07 10:40:01
pico2.byu.edu	red1.byu.edu	TCP	4222	1367	2015-05-20 15:00:19	2015-05-07 10:41:44
pico3.byu.edu	dm01client02-vip.byu.edu	TCP	1521	563	2015-05-20 10:13:21	2015-05-07 10:36:08
Source	Destination	Proto	Dst Port	Count	Last Seen	First Seen

Showing 1 to 10 of 17 entries

Previous 1 2 Next

Graph

Figure 4-2: Web Application Interface

A Python network creation package named NetworkX performed the graph generation. It constructed an in-memory directed graph of all nodes and edges in a given query and exported that graph to a JSON edgelist that can be read by D3.js.

The SVG graph in the web interface was built using the D3.js visualization framework. It used the force directed graph type and supported animations and mathematical constraints on the nodes that allowed them to arrange themselves according to the number and arrangement of edges.

The table was built using the JavaScript framework DataTables. It supported live text filtering, sorting, and pagination that made it easy for the user to find the data they need.

The web application forms submitted their queries to Python CGI scripts that assembled SQL queries from the data in the web forms and returned the appropriate results to the web client via JSON.

Apache was configured to allow HTML and CGI access, and for security reasons the test environment was protected by an implementation of Central Authentication Service (CAS) single sign-on using the mod_cas plugin that restricts access to specific directory groups.

5 RESULTS AND DATA ANALYSIS

5.1 Functionality

ServiceMap provided basic functionality for a user to follow one of two learning methods. The first was when a user knows what they want to look for and wanted to validate that what they expect is actually occurring. The user could quickly find this information by pulling up specific hosts and service ports in the web interface and using the text filter on the table to validate that the transactions they expected to be happening were listed. Second, if a user was not sure what services a host was running or what other hosts it connected to regularly, they could use ServiceMap in an exploratory manner and select a few things at a time to get easily digestible information in the graph.

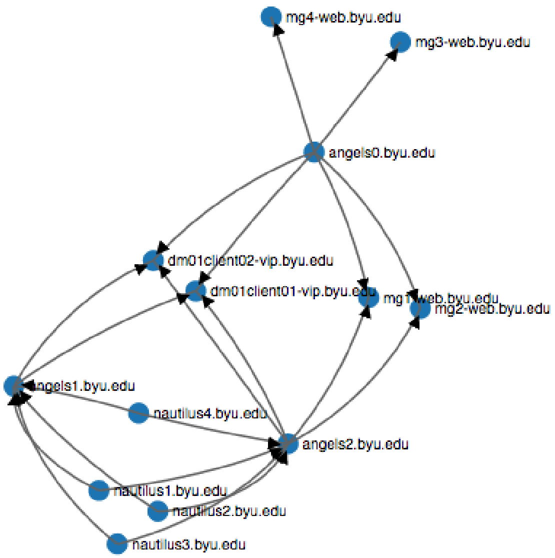


Figure 5-1: Sample Graph

5.2 Performance

5.2.1 CPU and Memory Utilization

The requirements were very low for both the agent systems and the collector. On the agents, the performance impact was negligible and so small as to be not easily measureable. The collector, which in the final configuration was receiving data from 573 agents, also had excellent performance.

Table 2: System Specification and Utilization

System	CPU Spec	CPU Use	RAM Spec	RAM Use	Net Spec	Net Use
Collector	2x Xeon E5-2670 cores	3% constant 55% summary job	1.8GB	71%	1Gbps	.03%
Agent	2x Xeon E5-2670 cores	0%	2GB	0%	1Gbps	0%

The hourly summary job ran in approximately three minutes on the collector system, and processed an average of 375,000 records each hour.

All three original objectives were achieved, proving the three hypotheses. Connections were successfully reported to a central collector (O1), a collector was created to store and sort data (O2), and a web application was created that provides useful views into the topology (O3). The three hypotheses were also proved, that it was possible to create an extensible model to describe dependency relationships between computer systems (H1), that systems could do some level of automatic dependency/interaction reporting on layer 4 to build relationship models programmatically (H2), and that topology maps assembled from system-reported data would give a useful and accurate picture of the environment (H3).

5.2.2 Network Bandwidth

Due to the compact nature of the syslog messages sent over the network and the rate limiting in the iptables rules to 20 messages per minute, overall network utilization was minimal. With 573 hosts reporting to the collector, the total incoming bandwidth averaged 300 kilobits/second. The tool did not affect outgoing bandwidth on the collector, as the incoming traffic is all UDP with no expected response.

5.2.3 Database Storage Requirements

The storage requirements were also quite small. While the volume of messages collected over a month-long period totaled several terabytes, the hourly summary job compressed that volume to just 1.3 million rows in the summary table. This is because most of the messages were repeating connections, so a counter was incremented on an existing database row instead of adding a new entry. With the 1.3 million unique rows in the summary table, storage usage was just 8 gigabytes.

5.3 Validity

Validity is an important metric for this method of gathering data. In order to keep the network bandwidth and CPU resources to a minimum, the agents were rate limited to 20 messages per minute. This means that not every single connection was logged at each occurrence. However, because the tool's primary focus was providing a view of what connections are used on a system and not an exact snapshot in time, it did not take long before all connections that were used programmatically are logged.

In order to further increase the signal to noise ratio in ServiceMap, any connections that had not been logged at least 50 times before were not shown in the web interface. Experience showed that within a few hours on a typical system, all connections that are essential to its business function had been logged and exceeded that threshold. In exceptional cases that time to 100% accuracy took as long as two weeks due to very long-lived connections not reaching the threshold of 50 repeated entries.

5.3.1 Manual Connection Discovery Comparison

To verify the validity of the data, a random set of 20 systems was chosen for manual examination and subsequent comparison with the data in the tool. The data was manually gathered by running the command “netstat -a --numeric-ports | grep -i established” to create a listing of all established layer 4 connections. The netstat tool is the standard used by system administrators for exposing data about connections, but because it simply queries the kernel’s table of current connections when it is run, it cannot show connections that do not exist at the moment it is run and will not often show short-lived connections.

That list was then compared with the information presented in ServiceMap. It was evident by comparing the number of unique connections discovered manually (a single snapshot in time when the command was run) to the number of unique connections reported in ServiceMap that while the common methods of manual connection discovery accurately represent a single snapshot in time, it was far more convenient to use the new application when determining the dependencies of a system. ServiceMap provided a valid view of those dependencies and provided more information than the standard method of manual discovery.

Table 3: Manual Accuracy Verification

Hostname	Percentage of manually discovered connections that were also listed in application	Number of manually discovered connections	Number of application reported connections
angels1	100%	6	24
bilbo1	100%	53	270
bosshoss1	100%	7	12
buggs1	100%	11	76
catapult1	100%	8	16
charge3	100%	5	13
chewy1	100%	11	41
chewy7	100%	9	44
daisy1	100%	7	37
donald1	100%	13	43
entropy1	100%	10	23
euro1	100%	9	43
legolas2	100%	9	34
month1	100%	5	15
mustket1	100%	7	15
neo1	100%	7	10
peabody1	100%	7	27
pico1	100%	9	18
rand1	100%	6	18
sticks1	100%	4	20

5.3.2 Alternative Tool Comparison

As seen by the results of the manual validity verification, the data gathered by the tool was considered 100% valid after a significant period of time to gather data. That time period depends on the nature of the business processes run by the host and if they are used frequently or seldom.

Like ServiceNow Discovery and JDisc Dependency Mapping, tools that use SNMP or remote logins to gather data, ServiceMap had little network or CPU impact. However, those tools would not be as confident in the validity of the data since their data is always a composed of instant snapshots in time, each of which is equally likely to miss short-lived connections such as DNS queries. The SolarWinds Orion tool would be 100% valid by capturing all data, but would also require expensive network hardware and storage resources to do so. The cost to operate that type of product would increase greatly with the number of systems supported. ServiceMap scaled well (the reference collector platform did not need increased resources to handle the full test server load) and agents could be easily split between multiple collectors if necessary for performance.

Table 4: Product Accuracy and Cost

Product	**Accuracy**	**Cost (Monetary and Time)**
ServiceNow Discovery	Snapshot in time	Medium
JDisc Dependency Mapping	Snapshot in time	Medium
SolarWinds Orion	100%	High
ServiceMap	Approaches 100% over time	Low

Compared to the market alternatives as described by their marketing and information sheets, ServiceMap was easy to deploy, consumed few resources, and was effectively 100% accurate beyond a window of a few days to a week.

5.4 **Utility**

ServiceMap proved to be very useful to a number of engineers. The combination of the table view and the graph view made it useful for a variety of tasks. The graph was only really useful when displaying fewer than a dozen or so nodes, but was very useful for spotting inconsistencies in a clustered system. There were several instances when the tool showed that one cluster member out of four was not communicating with a host that the other cluster members were all communicating with. This was helpful for proactive troubleshooting as there was not a way to easily see that information otherwise.

The table was also useful to see a representative count of a given connection, and it helped highlight several instances where a cluster was not evenly load balanced and one member was making the majority of the connections.

Further benefits were evident when talking with users outside of the core target group of system administrators. System administrators commonly had elevated access to all of the systems they are interested in and training on how to perform manual connection discovery. However, to a system administrator who did not possess all of those things or to another IT professional, this tool could be used to provide easy access to information that would have been very difficult to obtain otherwise.

5.4.1 Survey Results

A survey was distributed to gather feedback about how ServiceMap would be used.

The first question asked how likely the user was to use servicemap.byu.edu for change management tasks. On a scale of 1-10, the mean value was 5.5. This illustrates that ServiceMap may not be as intuitive for change management tasks, or that the engineers simply feel familiar enough with their systems to not need additional information before making a change.

The second question asked how likely the user was to use servicemap.byu.edu for troubleshooting tasks. On a scale of 1-10, the mean value was 7.7. This is much higher than the use for change management tasks, and likely shows the true value of ServiceMap. It is an informational tool, and people are generally not likely to seek out this type of information until they encounter a problem.

The third question asked how likely the user was to use servicemap.byu.edu for general learning or informational tasks. On a scale of 1-10, the mean value was 6.9. Anecdotally, this may be higher than ServiceMap's long-term use because right now it is novel and is something the engineers haven't seen before. Once they have satisfied their curiosity about the particular systems they care about, this type of usage may diminish.

The fourth question asked how often the user did use or expected to use servicemap.byu.edu. The most common response was 2-3 times per month, with a couple of users expecting to use it infrequently and a handful of users expecting to use it weekly or more.

A form was also provided for survey respondents to provide any other feedback or suggestions, some of which are discussed in Chapter 6.

5.5 **Costs Versus Benefits**

ServiceMap had a high benefit to cost ratio. While the time spent in development was significant, the stable nature of the open source components of ServiceMap ensure that it will continue to function for many years with little attention. The low resource utilization of both the collector and agents ensured little financial impact to an enterprise environment, while the simple deployment of both agent and collector components put little load on system administrators.

6 CONCLUSIONS AND RECOMMENDATIONS

6.1 Conclusions

The original aim of this research was to prove the utility of an approach for providing users with information about how systems are communicating in order to improve change management. The example tool succeeds at providing information in a way that was not easily possible previously. However, it appears that the primary reason to seek for such information is not change management (where users already assume they know all they will need) but rather for troubleshooting. When diagnosing problems, the tool is now a first-line method of information discovery for those familiar with it. While not the exact outcome envisioned at the beginning, it is at least as helpful and worth continued refinement and deployment.

6.2 Future Work

As the tool was distributed and used by dozens of engineers and IT professionals after sufficient completion, a number of suggestions and requests for enhancement were received. A few of these were not feasible due to the nature of the current method of data collection, but others warrant spending the time to implement them.

Improved User Interface: Several requests were received to add options to the user interface to configure the existing defaults to suit the needs of particular users. From one tester, “Time frame searching. That way I can compare traffic before and after a change. As we already talked about, add an option to not limit the output to 50+ connections on an individual port if desired. Perhaps combine that with the above suggestion in an "advanced search options" area.” For example, when examining a system that maintains long-lived connections, the threshold of logging 50 repeated connections before displaying in the web interface is not helpful. Other defaults that exist are excluded networks such as those outside of the scope of the tool, for which a configuration may also be added.

A number of improvements have also been requested to the graph part of the interface. The D3.js library used supports many more features that may be implemented to provide more visual information on the graph while also accommodating larger data sets by dynamically scaling and zooming, etc.

Increased Frequency of Summary Job: When users are testing changes they've made, usually as part of a fix to an existing problem, it would be helpful to not have to wait until the next hour rolls over to see the results. This could be fixed by simply increasing the frequency of the summary job, which would put more load on the collector (but load is currently very low) or by adding a button to run an on-demand summary job at the user's request.

Mapping of Service Ports to Service Names: The services that agents connect to are currently displayed only by port number. There are premade databases that associate well-known ports to service names, but these are usually highly inaccurate in an enterprise environment where many applications will be running on non-standard ports. It is also possible to have multiple systems utilizing the same port number for different purposes. A reasonable solution to this problem may be to provide a form in the web application for users to submit their own human-readable names for numeric port/protocol combinations. These names could then be displayed alongside the port numbers in the interface to assist users who may not be familiar with every service the system provides or consumes.

Granular Access Controls: As developed, ServiceMap itself does not provide any access controls, and any user who has access to the web interface may access all of its information. It may be helpful to provide a mechanism to limit users to view only systems pertinent to their duties. As currently deployed, the system uses an authentication service to allow or deny full access.

Reporting Framework and Monitoring Integration: A reporting framework could be developed to provide alerts to users or a monitoring solution such as Nagios when certain conditions were met. One tester wrote, “nagios alert integration. It would be nice to show alert relationships when multiple alerts occur at the same time.” This could help proactively prevent problems or simply speed troubleshooting by not requiring users to visit ServiceMap for certain types of information.

Increased Help and Training Materials: Several of the users expressed interest in ServiceMap but lacked understanding of how to use it efficiently or how to use the information to assist with their regular duties. One wrote, “I would appreciate some training or use cases that would help me understand how to utilize this tool with my daily tasks.” An article was written that describes how the tool works in an effort to promote understanding, but further training could be developed, specifically on how to use the features already present in the interface.

Integration with Application and Physical Network Topologies: This type of expansion is perhaps the most meaningful, and also very difficult. A user responded, “Great start. This needs to extend to all servers and all layers of connectivity so that you get to ServiceNow ServiceWatch type of expectations. What you really want is to be able to model a service and see all of the nodes and arcs that make up that service.” This would be a fitting research topic for a thesis or doctorate. Integrating the data gathered by this application into graphs of the physical network at OSI layers two or three would be reasonable and helpful. Integrating this data with application flow and business logic at OSI layer 7 would be incredible and provide information not currently accessible in all but a few isolated cases. As discussed in the literature review, the only reliable existing method of gathering layer 7 data is by instrumenting each specific application. However, if a method were developed to trace a client computer's request across, for example, a reverse proxy, web server, application server, database server, and back, it would be invaluable.

REFERENCES

AdRem. 2015. “AdRem NetCrunch.” Accessed June 16. http://www.adremsoft.com/netcrunch/.

AppNeta. 2014. “Application and Network Performance Management Tools.” Accessed April 22. http://www.appneta.com/products/.

Bahl, Paramvir, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. 2007. “Towards Highly Reliable Enterprise Network Services via Inference of Multi-Level Dependencies.” *ACM SIGCOMM Computer Communication Review* 37 (4) (October 1): 13. doi:10.1145/1282427.1282383. http://dl.acm.org/citation.cfm?id=1282427.1282383.

Barham, Paul, and Richard Black. 2008. “Constellation: Automated Discovery of Service and Host Dependencies in Networked Systems.” *... , MSR-TR-2008-67*. http://research.microsoft.com/en-us/people/risaacs/constellationv2.pdf.

Basu, Sujoy, Fabio Casati, and Florian Daniel. 2008. “Toward Web Service Dependency Discovery for SOA Management.” *2008 IEEE International Conference on Services Computing* (July): 422–429. doi:10.1109/SCC.2008.45. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4578551.

Binz, Tobias, Uwe Breitenbucher, Oliver Kopp, and Frank Leymann. 2013. “Automated Discovery and Maintenance of Enterprise Topology Graphs.” In *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*, 126–134. Ieee. doi:10.1109/SOCA.2013.29. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6717295.

Chen, Xu, Ming Zhang, Z. Morley Mao, and Paramvir Bahl. 2008. “Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions.” In *8th USENIX Symposium on Operating Systems Design and Implementation*, 117–130. USENIX. https://www.usenix.org/legacy/event/osdi08/tech/full_papers/chen_xu/chen_xu_html/.

Docusnap. 2015. “Docusnap.” Accessed June 16. http://www.docusnap.com/en.

Hewlett-Packard. 2015. “HP OpenView.” Accessed June 16. https://h20392.www2.hp.com/portal/swdepot/displayProductInfo.do?productNumber=PERFMINFO.

JDisc. 2014. “JDisc Dependency Mapping for JDisc Discovery Inventory Solution.” Accessed May 19. http://www.jdisc.com/en/products/dependencymapping.

Lumeta. 2015. “Lumeta IPSonar.” Accessed June 16. http://www.lumeta.com/product/ipsonar.html.

Nmap. 2015. “Nmap.” Accessed June 16. http://nmap.org/.

PacketTrap. 2015. “PacketTrap.” Accessed June 16. http://www.packettrap.com/.

Scrutinizer. 2015. “Scrutinizer.” Accessed June 16. https://www.plixer.com/Scrutinizer-Netflow-Sflow/scrutinizer.html.

ServiceNow. 2014. “ServiceNow Discovery.” Accessed May 19. http://www.servicenow.com/content/dam/servicenow/documents/datasheets/ds-discovery-20130103.pdf.

Solarwinds. 2014. “Solarwinds Log & Event Manager.” Accessed August 14. http://www.solarwinds.com/log-event-manager.aspx.

Spiceworks. 2014. “Network Monitoring and IT Management.” Accessed April 22. http://www.spiceworks.com/.

Tenable. 2014. “Tenable Log Correlation Engine.” Accessed August 14. http://www.tenable.com/products/log-correlation-engine.

VMware. 2014. “VMware vCenter Application Discovery Manager.” Accessed May 19. https://www.vmware.com/pdf/vcenter-application-discovery-manager-700-administration-guide.pdf.

WhatsUp Gold. 2014. “WhatsUp Gold.” Accessed April 19. http://www.whatsupgold.com/glossary/network-device-discovery/layer-2-3-discovery-tool.aspx.

Zimmermann, Hubert. 1980. “OSI Reference Model--The ISO Model of Architecture for Open Systems Interconnection.” *IEEE Transactions on Communications* 28 (4) (April): 425–432. doi:10.1109/TCOM.1980.1094702. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1094702.

APPENDICES

APPENDIX A. SOURCE CODE FOR LINUX AGENT

Iptables Configuration

```
# insert into the filter table
# clawsonj: added for logging for mapping project
:TOPOLOGY-LOG - [0:0]
-A TOPOLOGY-LOG -m mark --mark 0x6 -j RETURN
-A TOPOLOGY-LOG -j LOG --log-prefix "newConn " --log-level info
-A TOPOLOGY-LOG -j MARK --set-mark 0x6

-I INPUT -m state --state ESTABLISHED -m limit --limit 20/minute -j
TOPOLOGY-LOG
```

RSYSLOG Rule File

```
# /etc/rsyslog.d/iptables.conf
# write messages to mysql database
:msg, contains, "newConn" @bench1.byu.edu
& ~
```

APPENDIX B. SOURCE CODE FOR COLLECTOR

RSYSLOG Configuration

```
# requires rsyslog-mysql package
$ModLoad imudp
$UDPServerRun 514
# write messages to mysql database
$ModLoad ommysql.so
:msg,contains, "newConn"
:ommysql:localhost,Syslog,databaseuser,databasepassword
& ~
```

Database Create Scripts

Run the “createDB.sql” script included with the RSYSLOG installation to create the Syslog database and default tables.

```
CREATE TABLE LogSummary(
      id int auto_increment,
      srcip varchar(15),
      dstip varchar(15),
      proto varchar(5),
      dstport smallint(6),
      count int,
      lastseen datetime,
      firstseen datetime,
      PRIMARY KEY (`id`,`srcip`,`dstip`,`proto`,`dstport`)
      );
```

Hourly Summary Job

```
#!/usr/bin/env python

# connectionSummary.py
# symlink to this file goes in /etc/cron.hourly

# log into mysql database, read from rsyslog's SystemEvents table,
# summarize data into another table, then delete the rows we
summarized from SystemEvents.
from __future__ import print_function
from decimal import Decimal
from datetime import datetime, date, timedelta
import mysql.connector
```

```
# PROPERTIES
counter = 0
debug = False

# Connect with the MySQL Server
cnx = mysql.connector.connect(user='databaseuser', database='Syslog',
password='databasepassword')

# Get two buffered pcursors
curA = cnx.cursor(buffered=True)
curB = cnx.cursor(buffered=True)
curC = cnx.cursor(buffered=True)

# Query to get the data we care about sent by iptables logging to
syslog
get_data = (
    "SELECT ID, FromHost, DeviceReportedTime, Message FROM
SystemEvents "
    "where Message like '%newConn%' AND Message not like '%127.0.0.1%'
order by DeviceReportedTime limit 500000")

delete_data = ("delete from SystemEvents where ID = \'%s\'")
delete_all_data = ("delete from SystemEvents")

# get the data
curA.execute(get_data)

# UPDATE and INSERT statements for the LogSummary table
insert_log_summary = (
    "INSERT INTO LogSummary (srcip, dstip, proto, srcport, dstport,
counter, lastseen, firstseen) "
    "VALUES (%s, %s, %s, %s, %s, %s, %s, %s) ON DUPLICATE KEY UPDATE
counter=counter+1, lastseen=VALUES(lastseen)")

# iterate through results and do string parsing on the text message
reversed = 0
reversed2049 = 0

for (ID, FromHost, DeviceReportedTime, Message) in curA:
    messageList = Message.split(" ")
    messageDict = {}

    # build dictionary from valid parts of iptables message
    for index, item in enumerate(messageList):
        if "=" in item:
            parts = item.split("=")
            if parts[1]:
                messageDict[parts[0]] = parts[1]

    try:
        if 'DPT' in messageDict:
            # convert strings to ints so we can compare them
            messageDict['DPT'] = int(messageDict['DPT'])
            messageDict['SPT'] = int(messageDict['SPT'])

            if messageDict['DPT'] <= messageDict['SPT'] and
messageDict['SPT'] != 2049:
                if debug and messageDict['SPT'] == 2049:
                    print("normal destination ", messageDict['DPT'],
"; source ", messageDict['SPT'])
                sourcePort = messageDict['SPT']
                destinationPort = messageDict['DPT']
                sourceHost = messageDict['SRC']
                destinationHost = messageDict['DST']

            elif messageDict['DPT'] != 2049:
                sourcePort = messageDict['DPT']
                destinationPort = messageDict['SPT']
                sourceHost = messageDict['DST']
                destinationHost = messageDict['SRC']
                reversed += 1
                if debug:
                    print("reversed destination ", messageDict['DPT'],
" to source and source ", messageDict['SPT'] ," to destination.")

            curB.execute(insert_log_summary,
                #(messageDict['SRC'], messageDict['DST'],
messageDict['PROTO'], "0", messageDict['DPT'], "1",
DeviceReportedTime, DeviceReportedTime))
                (sourceHost, destinationHost, messageDict['PROTO'],
sourcePort, destinationPort, "1", DeviceReportedTime,
DeviceReportedTime))

        #curC.execute(delete_data, (ID))
        curC.execute(delete_all_data)
        counter += 1
    except mysql.connector.Error as err:
        print("Had a problem with the database: {}".format(err))

print("Processed records:", counter)
if debug:
    print("Reversed source/destination:", reversed)
    print("Reversed 2049 source/destination:", reversed2049)

cnx.commit()
cnx.close()
exit()
```

Web Application HTML

This HTML and JavaScript code is just the core application and does not include any security or template styling.

```
<!-- /var/www/html/index.html -->
<head>
<!-- JS links and CSS for dataTables and JQuery -->
    <script src="https://code.jquery.com/jquery-
1.11.2.min.js"></script>
    <link rel="stylesheet"
href="https://cdn.datatables.net/1.10.5/css/jquery.dataTables.css" />
     <script
src="https://cdn.datatables.net/1.10.5/js/jquery.dataTables.min.js"></
script>
    <style>
        path.link {
          fill: none;
          stroke: #666;
          stroke-width: 1.5px;
        }

        circle {
          fill: #ccc;
          stroke: #fff;
          stroke-width: 1.5px;
        }

        text {
          fill: #000;
          font: 10px sans-serif;
          pointer-events: none;
        }

     </style>
</head>

<body>

<div id="selectors">
<div id="hostDiv" style="float:left;">
<h3>Hosts</h3>
    <form name="hostForm" method="post">
        <select id="hostList" name="hostList" multiple size="20"
required style="width:350px;">
        </select>
        <input type="submit" name="hostSubmit" class="button"
id="hostSubmit" value="Go"/>
    </form>
</div>
```

```
<div id="portDiv" style="float:left;padding-left:20px">
<h3>Service Ports</h3>
    <form name="portForm" method="post">
        <select id="portList" name="portList" multiple size="20"
required  style="width:100px;">
        </select>
        <input type="submit" name="portSubmit" class="button"
id="portSubmit" value="Go"/>
    </form>
</div>

</div>

<div style="clear:both;">&nbsp;</div>

<div id="tableDiv" style=>
    <h3>Results</h3>
    Note: The search box on the table does live filtering of the data.
    <table id="resultTable" class="display" cellspacing="0"
width="100%">
    <thead>
        <tr>
            <th>Source</th>
            <th>Destination</th>
            <th>Proto</th>
            <th>Dst Port</th>
            <th>Count</th>
            <th>Last Seen</th>
            <th>First Seen</th>
        </tr>
    </thead>
    <tbody id="resultBody">
    </tbody>
    <tfoot>
        <tr>
            <th>Source</th>
            <th>Destination</th>
            <th>Proto</th>
            <th>Dst Port</th>
            <th>Count</th>
            <th>Last Seen</th>
            <th>First Seen</th>
        </tr>
    </tfoot>
    </table>
    <script> var dt = $('#resultTable').dataTable(); </script>
</div>
<br>
<div id="graphDiv">
<h3>Graph</h3>
</div>
<script>
```

```
// load host list into hostForm
$.ajax({
    url:'cgi-bin/hostList.py',
    type:'POST',
    dataType: 'json',
    success: function( json ) {
        $('#hostList').empty();
        $.each(json, function(key, value) {
            $('#hostList').append($('<option>').text(key + " (" +
value + ")").attr('value', value));
        });
    }
});

// load port list from host selection
$(function() {
    $("#hostSubmit").click(function() {
        // validate and process form here
        dataString = "";
        var hosts = $("select#hostList").val();
        for (var i = 0; i < hosts.length; i++) {
            dataString += "hostList=" + hosts[i] + "&";
        }
        //alert (dataString); return false;
        $.ajax({
            url:'cgi-bin/portList.py',
            data: dataString,
            type:'POST',
            dataType: 'json',
            success: function( json ) {
                $('#portList').empty();
                $.each(json, function(key, value) {

$('#portList').append($('<option>').text(value).attr('value', value));
                });
            }
        });
        return false;
    });
});
$(function() {
    $("#portSubmit").click(function() {
        // validate and process form here
        dataString = "";
        var ports = $("select#portList").val();
        for (var i = 0; i < ports.length; i++) {
            dataString += "portList=" + ports[i] + "&";
        }
        var hosts = $("select#hostList").val();
        for (var i = 0; i < hosts.length; i++) {
            dataString += "hostList=" + hosts[i] + "&";
        }
```

```
//alert (dataString); return false;
$.ajax({
    url:'cgi-bin/connectionList.py',
    data: dataString,
    type:'POST',
    dataType: 'json',
    success: function( json ) {
        // clear table of old data
        var dt = $('#resultTable').DataTable();
        dt
            .clear()
            .search('')
            .columns().search('');
        // load new data into table
        $.each(json['list'], function(key, value) {
            var counter = "";
            var dstip = "";
            var dstport = "";
            var firstseen = "";
            var lastseen = "";
            var proto = "";
            var srcip = "";

            $.each(value, function(key, value) {
                switch (key)
                {
                case "counter":
                    counter = value;
                    break;
                case "dstip":
                    dstip = value;
                    break;
                case "dstport":
                    dstport = value;
                    break;
                case "firstseen":
                    firstseen = value;
                    break;
                case "lastseen":
                    lastseen = value;
                    break;
                case "proto":
                    proto = value;
                    break;
                case "srcip":
                    srcip = value;
                    break;
                }
            });

            dt.row.add( [
                srcip,
                        dstip,
                        proto,
                        dstport,
                        counter,
                        lastseen,
                        firstseen
                    ] );
                    dt.draw();
                });

                // call graphing function
                buildGraph(json['graph']);

            }
        });
        return false;
    });
});

</script>


<div style="clear:both;">&nbsp;</div>

<!-- BEGIN D3.js stuff -->
<script src="/static/d3.v3.min.js" charset="utf-8"></script>

<script>

    function buildGraph(jsonData) {
        var width = 960,
            height = 800;

        var color = d3.scale.category20();

        var force = d3.layout.force()
            .charge(-400)
            .linkDistance(30)
            .linkStrength(3)
            .size([width, height]);

        d3.select("svg").remove();
        var svg = d3.select("#graphDiv").append("svg")
            .attr("width", width)
            .attr("height", height);

          // create arrowhead marker end
          svg.append("defs").selectAll("marker")
            .data("arrowhead")
            .enter().append("marker")
            .attr("id", "arrowhead")
            .attr("viewBox", "0 -5 10 10")
            .attr("refX", 15)
            .attr("refY", -1.5)
            .attr("markerWidth", 6)
            .attr("markerHeight", 6)
            .attr("orient", "auto")
          .append("path")
            .attr("d", "M0,-5L10,0L0,5");

        graph = jsonData;
          var nodes = graph.nodes.slice(),
              links = [],
              bilinks = [];

          graph.links.forEach(function(link) {
            var s = nodes[link.source],
                t = nodes[link.target],
                i = {}; // intermediate node
            nodes.push(i);
            links.push({source: s, target: i}, {source: i, target:
t});
            bilinks.push([s, i, t]);
          });

          force
              .nodes(nodes)
              .links(links)
              .start();

          var node = svg.selectAll(".node")
              .data(graph.nodes)
              .enter().append("circle")
              .attr("class", "node")
              .attr("r", 8)
              .style("fill", function(d) { return color(d.group); })
              .call(force.drag);

          var link = svg.selectAll(".link")
              .data(bilinks)
              .enter().append("path")
              .attr("class", "link")
              .attr("marker-end", "url(#arrowhead)");

        var text = svg.append("g").selectAll("text")
            .data(force.nodes())
            .enter().append("text")
            .attr("x", 10)
            .attr("y", ".35em")
            .text(function(d) { return d.hostname; });

          force.on("tick", function() {
            link.attr("d", function(d) {
              return "M" + d[0].x + "," + d[0].y
```

```
                  + "S" + d[1].x + "," + d[1].y
                  + " " + d[2].x + "," + d[2].y;
            });
            node.attr("transform", function(d) {
              return "translate(" + d.x + "," + d.y + ")";
            });
            text.attr("transform", function(d) {
              return "translate(" + d.x + "," + d.y + ")";
            });
          });
    }
</script>
<!-- END D3.js stuff -->
</body>
```

Web Application CGI Host List

This program returns a JSON-formatted list of all hosts that match specific networks.

```
#!/usr/bin/env python
# /var/www/cgi-bin/hostList.cgi

# log into mysql database, get list of hosts, write to JSON.

from __future__ import print_function
from decimal import Decimal
from datetime import datetime, date, timedelta
import mysql.connector
from socket import gethostbyaddr
from socket import gethostbyname
import json
import sys
import collections


def nslooky(ip):
    try:
        output = gethostbyaddr(ip)
        return output[0]
    except:
        output = ip
        return output

def getip(hostname):
    try:
        output = gethostbyname(hostname)
        return output
    except:
        return "hostname not found."
```

```
def main(argv):

    # PROPERTIES
    debug = False

    # Connect with the MySQL Server
    cnx = mysql.connector.connect(user='databaseuser',
database='Syslog', password='databasepassword')

    # Get buffered pcursor
    curA = cnx.cursor(buffered=True)

    # build list of desired subnets, in sql syntax
    subnets = [
            '192.168.0%', # Class C
            '10.%', # Class B
            # add as many as desired, or remove the subnetFilter from
the SQL select statement to see everything.
            ]

    sqlSeparator = "' or dstip like '"
    subnetFilter = sqlSeparator.join(subnets)

    get_data = ("select distinct(dstip) from LogSummary where (dstip
like '" + subnetFilter + "') and counter > 50 order by
INET_ATON(dstip)")

    # get the data
    curA.execute(get_data)
    cnx.commit()
    cnx.close()

    results = {}
    # iterate through results and build dictionary with hostname:ip
    for (dstip) in curA:
        addr = dstip[0]
        results[nslooky(addr)] = addr

    sortedResults = collections.OrderedDict(sorted(results.items()))
    print("Content-Type: application/json")
    print("")
    print(json.dumps(sortedResults))

if __name__ == "__main__":
    main(sys.argv[1:])
```

Web Application CGI Port List

This program returns a JSON-formatted list of all service ports are used by the selected set of hosts.

```
#!/usr/bin/env python
# /var/www/cgi-bin/portList.cgi

# log into mysql database, get list of hosts, write to JSON.

from __future__ import print_function
from decimal import Decimal
from datetime import datetime, date, timedelta
import mysql.connector
import json
import sys
import collections
import cgi
import cgitb; cgitb.enable()

def main():

    # PROPERTIES
    debug = False
    form = cgi.FieldStorage()

    # Connect with the MySQL Server
    cnx = mysql.connector.connect(user='databaseuser',
database='Syslog', password='databasepassword')

    # Get buffered pcursor
    curA = cnx.cursor(buffered=True)

    r = form.getlist('hostList')
    sqlSeparator = "' or dstip='"
    dstFilter = sqlSeparator.join(r)

    srcSeparator = "'or srcip='"
    srcFilter = srcSeparator.join(r)

    fields = "<p>"+ str(r) +"</p>"

    #get_data = ("select distinct(dstport) from LogSummary where
dstip='" + dstFilter + "' and counter > 50 order by dstport")
    get_data = ("select distinct(dstport) from LogSummary where
(dstip='" + dstFilter + "' or srcip='" + srcFilter + "')  and counter
> 50 order by dstport")

    # get the data
    curA.execute(get_data)
    cnx.commit()
    cnx.close()

    results = []
    # iterate through results and build dictionary with hostname:ip
    for (dstport) in curA:
        port = dstport[0]
        results.append(port)

    print("Content-Type: application/json")
    print("")
    print(json.dumps(results))

if __name__ == "__main__":
    main()
```

Web Application CGI Connection List

This program returns a JSON-formatted list of all connections associated with the selected hosts and ports and uses NetworkX to generate a graph and return a JSON-formatted edgelist to be displayed by D3.js in the browser.

```
#!/usr/bin/env python
# /var/www/cgi-bin/connectionList.cgi

# log into mysql database, get list of hosts, write to JSON.

from __future__ import print_function
from decimal import Decimal
from datetime import datetime, date, timedelta
import mysql.connector
import networkx as nx
from networkx.readwrite import json_graph
import json
import sys
import collections
import cgi
import cgitb; cgitb.enable()
from socket import gethostbyaddr

def nslooky(ip):
    try:
        output = gethostbyaddr(ip)
        return output[0]
    except:
        output = ip
        return output
```

```
def main():

    # PROPERTIES
    debug = False
    form = cgi.FieldStorage()

    # Connect with the MySQL Server
    cnx = mysql.connector.connect(user='databaseuser',
database='Syslog', password='databasepassword')

    # Get buffered pcursor
    curA = cnx.cursor(buffered=True)

    r = form.getlist('hostList')
    sqlSeparatorDst = "' OR dstip='"
    dstFilter = sqlSeparatorDst.join(r)

    sqlSeparatorSrc = "' OR srcip='"
    srcFilter = sqlSeparatorSrc.join(r)

    r2 = form.getlist('portList')
    sqlSeparatorPort = "' OR dstport='"
    portFilter = sqlSeparatorPort.join(r2)

    get_data = ("select srcip, dstip, proto, dstport, counter,
lastseen, firstseen from LogSummary where ((dstip='" + dstFilter + "'
OR srcip='" + srcFilter + "') AND (dstport='" + portFilter + "')) AND
counter > 50 order by INET_ATON(dstip)")


    # get the data
    curA.execute(get_data)
    cnx.commit()
    cnx.close()

    results = []
    G = nx.DiGraph()

    # iterate through results and build dictionary with hostname:ip
    for (srcip, dstip, proto, dstport, counter, lastseen, firstseen)
in curA:
        dict = {}
        dict["srcip"] = nslooky(srcip)
        dict["dstip"] = nslooky(dstip)
        dict["proto"] = proto
        dict["dstport"] = dstport
        dict["counter"] = counter
        dict["lastseen"] = str(lastseen)
        dict["firstseen"] = str(firstseen)
        results.append(dict)

        # now add node to graph
        G.add_node(srcip)
        G.node[srcip]['hostname'] = dict["srcip"]
        G.add_node(dstip)
        G.node[dstip]['hostname'] = dict["dstip"]
        G.add_edge(srcip, dstip)
        G.edge[srcip][dstip]['dstport'] = dstport
        G.edge[srcip][dstip]['counter'] = counter
        G.edge[srcip][dstip]['proto'] = proto



    print("Content-Type: application/json")
    print("")
    resultJson = json.loads(json.dumps(results))
    graphJson = json.loads(json.dumps(json_graph.node_link_data(G)))
    response = {"list": resultJson, "graph": graphJson}
    print(json.dumps(response))

if __name__ == "__main__":
    main()
```