Comparing Expected and Real-Time Spotify Service Topology

VILIUS VISOCKAS

KTH Information and Communication Technology

Degree project in
Communication Systems
Second level, 30.0 HEC
Stockholm, Sweden

MASTER OF SCIENCE THESIS

Comparing Expected and Real–Time Spotify Service Topology

Vilius VISOCKAS
vilius.visockas@gmail.com

May 29, 2012

Comparing Expected and Real–Time Spotify Service Topology

Final version (2012-05-30)

VILIUS VISOCKAS *(vilius.visockas@gmail.com)*
Master's Programme in Security and Mobile Computing
NordSecMob (KTH + NTNU track)

KTH Royal Institute of Technology
School of Information and Communication Technology
Stockholm, Sweden
academic supervisor Prof. Gerald Q. Maguire Jr. *(maguire@kth.se)*

NTNU Norwegian University of Science and Technology
Department of telematics
Trondheim, Norway
academic supervisor Prof. Yuming Jiang *(jiang@item.ntnu.no)*

Spotify AB
Stockholm, Sweden
industrial supervisor Mattias Jansson *(mattias.jansson@spotify.com)*

Abstract

Spotify is a music streaming service that allows users to listen to their favourite music. Due to the rapid growth in the number of users, the amount of processing that must be provided by the company's data centers is also growing. This growth in the data centers is necessary, despite the fact that much of the music content is actually sourced by other users based on a peer-to-peer model.

Spotify's backend (the infrastructure that Spotify operates to provide their music streaming service) consists of a number of different services, such as track search, storage, and others. As this infrastructure grows, some service may behave not as expected. Therefore it is important not only for Spotify's operations[1] team, but also for developers, to understand exactly how the various services are actually communicating.

The problem is challenging because of the scale of the backend network and its rate of growth. In addition, the company aims to grow and expects to expand both the number of users and the amount of content that is available. A steadily increasing feature-set and support of additional platforms adds to the complexity. Another major challenge is to create tools which are useful to the operations team by providing information in a readily comprehensible way and hopefully integrating these tools into their daily routine. The ultimate goal is to design, develop, implement, and evaluate a tool which would help the operations team (and developers) to understand the behavior

[1]Also known as the Service Reliability Engineers Team (SRE).

of the services that are deployed on Spotify's backend network.

The most critical information is to alert the operations staff when services are not operating as expected. Because different services are deployed on different servers the communication between these services is reflected in the network communication between these servers. In order to understand how the services are behaving when there are potentially many thousands of servers we will look for the patterns in the topology of this communication, rather than looking at the individual servers. This thesis describes the tools that successfully extract these patterns in the topology and compares them to the expected behavior.

Sammanfattning

Spotify är en växande musikströmningstjänst som möjliggör för dess användare att lyssna på sin favoritmusik. Med ett snabbt växande användartal, följer en tillväxt i kapacitet som måste tillhandahållas genom deras datacenter. Denna växande kapacitet är nödvändig trots det faktum att mycket av deras innehåll hämtas från andra användare via en peer-to-peer modell.

Spotifys backend (den infrastruktur som kör Spotifys tjänster) består av ett antal distinkta typer som tillhandahåller bl.a. sökning och lagring. I takt med att deras backend växer, ökar risken att tjänster missköter sig. Därför är det inte bara viktigt för Spotifys driftgrupp, utan även för deras utvecklare, att förstå hur dessa kommunicerar.

Detta problem är en utmaning p.g.a. deras storskaliga infrastruktur, och blir större i takt med att den växer. Företaget strävar efter tillväxt och förväntar detta i både antalet användare och tillgängligt innehåll. Stadigt ökande funktioner och antalet distinkta plattformar bidrar till komplexitet. Ytterligare en utmaning är att bidra med verktyg som kan användas av driftgrupp för att tillhandahålla information i ett tillgängligt och överskådligt format, och att förhoppningsvis integrera dessa i en daglig arbetsrutin.

Det slutgiltiga målet är att designa, utveckla, implementera och utvärdera ett verktyg som låter deras driftgrupp (och utvecklare) förstå beteenden i olika tjänster som finns i Spotifys infrastruktur. Då dessa tjänster är utplacerade på olika servrar, reflekteras kommunikationen mellan dem i deras nätverketskommunikation. För att förstå tjänsternas beteende när det potentiellt kan finnas tusentals servrar bör vi leta efter mönster i topologin, istället för beteenden på individuella servrar.

Acknowledgements

I want to thank my supervisor Mattias Jansson, who provided me with a good working environment at Spotify and was my first point of contact for everything involving this thesis project. In addition, it was an exciting experience to be involved in the operations and infrastructure automation teams and to come to understand what day-to-day network monitoring and operations feels like.

I would like to thank my main academic supervisor prof. Gerald Q. Maguire from KTH for his guidance during this thesis project. He deserves the most credits for helping me to improve my report and focusing my attention on important details. I am also grateful Yuming Jiang for being be my academic supervisor at NTNU.

Big thanks for my colleagues John-John Tedro and Martin Parm, who gave me presentation technique tips and the usage of various development tools. They were great table football partners during lunch time as well.

Contents

List of Tables

List of Figures

List of Acronyms

The following is the list of acronyms which are used in this thesis.

AP	Access Point for Spotify clients
BNF	Backus Naur Form
BPF	Berkeley Packet Filter
DDoS	Distributed Denial of Service attack
DNS	Domain Name System
IP	Internet Protocol
JSON	JavaScript Object Notation (data serialization format)
NetFlow	A network protocol for collecting IP traffic information
PCAP	Packet capture library
RFC	Request for Comments
SILK	System for Internet-Level Knowledge
SRE	Service Reliability Engineering team
SRV	DNS server record type
TCP	Transmission Control Protocol
TOS	Type of Service

List of Definitions

The following list of definitions briefly explains frequently used concepts in this thesis. It is important to read this section carefully, as doing so will help the reader to understand the rest of the thesis.

Operations	Spotify team responsible for ensuring all services work (also known as SRE).
Service Dependency	Two intercommunicating services are dependent upon each other, as in a *TCP* connection, with packets flowing in both directions. The terms *service topology* and *service correlation* are used as synonyms in this thesis. Generally, a service dependency can be explained as a graph, in which services are represented as nodes and there exists an edge between two services when there is network communication between them. The weight of the edge reflects the volume of this communication.
Service	In the scope of this thesis project, servers having similar functions which handle and respond to clients' requests *provide a service*. Services usually have a specific business value which often could benefit Spotify customers or other services. Programs running Spotify services are usually designed and implemented by Spotify engineers. An example of a service is a *Search service* which allows customers to locate tracks. In contrast, the *SQL* database is conceptually also a service, but it is not considered a Spotify service in this thesis. Nevertheless, the definition of a *service* is flexible and can be adapted to suite the specific contexts.
Site	In Spotify's context, a site is a geographically distinct data center which has a common point of presence on the Internet.

Spotify	Music streaming service. The data captured from the Spotify infrastructure network will be used for analysis.

Chapter 1

Introduction

This chapter defines the area of research, sets boundaries for the thesis project and delimits our investigation. The chapter ends with a description of the structure of the remainder of the thesis.

1.1 Problem and Goals

Spotify deploys a service based model on their backend. This backend consists of a internal network of potentially thousands of servers. Various services interact with clients and between each other. However, due to constant growth, the actual communication patterns may deviate from the planned communication patterns. Understanding these communication patterns is key to providing good service to the users and using as input to plan for the scaling of the various services in the backend. This scaling process includes deciding how many instances of each type of server and service is needed, as well as determining the internal and external network requirements.

This thesis project began with trying to understand the various services and how they interact with each other. The project's goal is to design, implement, and evaluate tools which would help the operations team by presenting to them a picture of the current service topology (i.e., the actual pattern of communications between services) and to highlight how this differs from the expected topology. Additionally, geographical *site* information can be associated with a service and presented along with the topology information.

This depiction could be augmented to suit several use cases that would compare how the expected network flows differ from the actual flows. Examining cases when services interact between different sites may lead to manually suggested optimizations. Automating this detection process is left for future work.

The tools we want to create should provide the *SRE* team with information about runtime backend service network dependencies. The designed dependencies will be compared with the actual dependencies. The actual dependencies will be determined by network analysis and possibly supplemented with data from the servers themselves. This information can be used when trying to understand the effects of these dependencies during incidents when the systems are not providing the expected services or not providing them with the expected service quality. Therefore a dependency graph is valuable both for impact analysis (what other services could be affected when a service malfunctions) and cause analysis (to understand why service malfunctions by investigating dependent services). From an operations point of view it is desirable to highlight deviations of the actual services behaviors from the designed behaviours.

1.2 Limitations of Investigation

This investigation will focus on the transport layer as opposed to application layer protocols. The focus of this analysis is on TCP transport protocol, as all services that we are concerned with in this thesis project use TCP as their transport protocol. This thesis will describe the methodology that could be applied to achieve the goal of this thesis, however the description of detailed results will be limited or obfuscated as these details are proprietary to the company.

The discussion of the specific logical relations between services (semantics) is not included in this report. Report simply describes the actual communication patterns between services.

The reader of this report should be familiar with basic computer networking concepts, have a basic understanding of *DNS* behaviour, as well as transport and network layers as defined in the *OSI* model; therefore these topics are not covered in the background section. The reader may want to use the results of this report to perform a similar analysis on another organizations' networks.

1.3 Evaluation of Results

This thesis will introduce several possible data collection methods, but only a few of them are feasible and suitable for this thesis project. Two of the possible collection methods will be selected and compared with respect to their impact to server performance.

In addition, two service classification methods will be compared using criteria such as accuracy of the results and execution time.

1.4 Structure of the thesis

Chapter 1 provided an introduction to this thesis. Chapter 2 will cover necessary background information. It will overview the Spotify's product, explain the architecture of Spotify's services and the structure of the backend network.

Chapter 3 contains an analysis of the problem. First, section 3.1 discusses the definition of a service in the Spotify's context. Section 3.2 gives an overview of possible methods to observe network flows. The chapter continues by describing how to determine service behaviour from network flow information. Section 3.3 will provide a deeper analysis of the selected network observation approaches using NetFlow data. It discusses available tools, possible deployment scenarios, and problematic areas of the implementation. Section 3.4 focuses on the analysis of data provided from the tools described in previous section. It identifies the problem of a lack of a convenient framework for plotting customized visual plots from NetFlow data and the limitations of the available command line tools. This chapter then describes the details of the design and implementation of such a framework.

Chapter 4 uses a combination of data collection and our own analysis tools to understand the actual backend service behaviour.

Chapter 5 summarizes our analysis. It also contains a discussion, conclusions and suggests potential future work.

Chapter 2

Background

The aim of this chapter is to give the necessary background information about the Spotify organization, its product, and the architecture supporting the delivery of this service.

2.1 About Spotify

Spotify is a music streaming service. The service is offered in three versions: a free version with advertisements, unlimited, and a premium (pay-per-month) version. As of January 2012, Spotify had 10,000,000 registered users, 20% of which subscribed to the premium service [1].

Spotify (here refers to the company and companies[1]) is a rapidly growing organization. During this thesis project the main goal was to support this growth, thus the backend must support more customers and it must do so efficiently. Achieving this goal demands that the infrastructure work properly, while supporting increasing loads and being constantly upgraded. At the time of writing this thesis, Spotify deployed over 1000 servers running various services. These servers are located in several *sites* in Europe and the United States

[1]Spotify Sweden AB, Spotify Limited, Spotify France SAS, Spotify Spain S.L, Spotify Norway AS, Spotify Netherlands B.V., Spotify USA Inc., Spotify Finland Oy, and Spotify Denmark ApS currently form the overall business concern, but organization is constantly growing.

of America.

2.2 Spotify's Architecture

Spotify is a music streaming service offering low-latency access to a library of over 15 million music tracks. Streaming is performed by a combination of client-server access and a peer-to-peer protocol between clients.

Spotify is based on various services, mostly of which are written in Python. The core element of the system is the Access Point (AP) which is the first point of contact for each client. There are two major versions of the client: a mobile and a desktop version. The whole Spotify infrastructure heavily relies on Domain Name System (*DNS*) queries to locate servers that provides some specific service. Although this approach has both advantages and disadvantages, it is generally agreed that it is both fast and efficient. Details of DNS can be found in the Internet Engineering Task Force (IETF) standard *RFC 1035*[2] [2]. In particular, Spotify's DNS is configured to return a list of SRV records for a particular service query, as described in *RFC 2782* [3]. Each SRV record, among other attributes, returns a port and host name. The host name can be resolved separately to an IP address. This tuple uniquely describes the destination of the service provider. If a service is moved to another machine, then the DNS records must be updated accordingly. The DNS servers use relatively small time-to-live values for these entries in order to avoid potential problems associated with DNS caching.

Services usually utilize TCP as their transport protocol and different services utilize specific TCP port numbers. Above the transport protocol, i.e., at the application level, the protocol that is used is HTTP, but other protocols are used as well in some cases. It is assumed that communication between services facilitates exactly one TCP connection, because special cases as FTP connection (which utilizes two parallel connections) do not exist in Spotify's context. However, each working instance of connection to the service (for example, when connecting to database) may have a seperate connection.

[2]There are more RFCs covering the behaviour of DNS, but here we refer only to the basic of DNS that are covered in the RFC 1035.

2.3 Example of service dependencies

Let us describe a realistic scenario of a service oriented architecture. Let us assume that a new company is creating a webmail service whose operation is sponsored by advertisements (ads).

The company plans to operate several servers, each dedicated for a different purpose (i.e., a service). The *Web* server handles users' requests and manages customers' accounts. The *Ads* server picks the most suitable ad to show based on the user's information in a database. The *Database* server stores each user's account information. The *Mail* server sends, receives, and stores email messages. The *Spam* detection server analyses text in email, and filters out messages which are identified as spam.

These servers communicate with each other, thus creating service dependencies (as shown in figure 2.1). For example, the *Web* server and *Ads* server fetch data from the *Database* server (in this example MySQL). In addition, the *Web* server uses the *Mail* server to obtain email messages and show them via a web interface. The *Mail* server uses responses from the *Spam* detection server to filter out spam. Most of the servers resolves hostnames to the IP addresses using the local *DNS* server. Finally, the *Mail* server uses a regularly updated external blacklist service to eliminate e-mail to and from sites listed in blacklist.

It is important to note that usually two communicating services can be classified as a *client* and *server*. The *client* service initiates the request to the *server* service and expects a response in order to continue its operation. Therefore while the edges between services in the topology graph indicate communication in both directions, we have used an arrow on only one end of the edge to illustrate a client-server relationship (the arrow head points to the *server*).

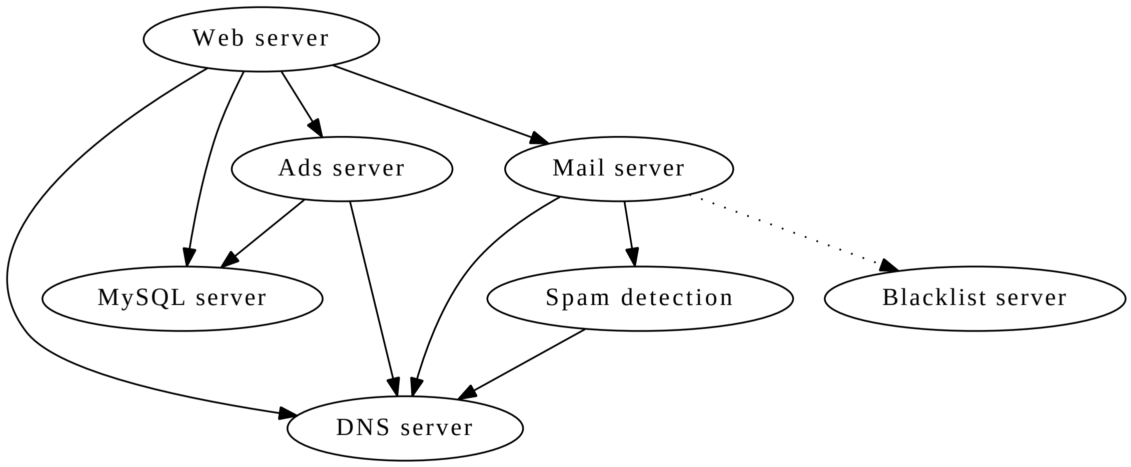

Figure 2.1: *Real life example of service dependencies*

A dependency graph such as this one helps us to understand the operation of the system. For example, if both *Web* server and *Ads* server start to fail to serve users' requests, it is highly likely that the *Database* server is malfunctioning, because it is used by both services.

This example suggests that services can be either *internal* (i.e., those which are in control of organization, such as *Mail* service), or *external* (such as the *Blacklist* service). In addition, services can be either *standard* (widely used, such MySQL server), or custom (such as the *Ads* service). Our focus in this thesis project is finding dependencies between *internal custom* Spotify services.

2.4 Relevant work

Attempts at understanding of live, automated backend network service dependencies was never done previously at Spotify. However, the general problem of service dependency is not new.

Ensel and Keller [4] suggest an approach for automated generation of service dependency models. The main idea is that measuring and correlating certain metrics (CPU load or bandwidth usage) on different hosts can infer a service dependency. In related work, these authors suggest managing service dependencies using XML and resource description framework [5]. The goal of such model is to understand impact of service failure.

A. Keller and G. Kam describe an architecture and its implementation for retrieving and handling dependency information from various managed resources in a web-based environment [6]. Their work relies on extracting information from system configuration repositories combined with results obtained from perturbing some components of a distributed system under a typical workload.

Basu, Fabio and Florian proposed a solution for the automatic identification of traces of dependent messages, based on the correlation of messages exchanged among services [7, 8]. Their key principle is inference of a causal dependency within message pairs in the log, where the first message is received at the service node from which the second message originates using a probabilistic model.

To summarize, there are several approaches for understanding and modelling service dependencies. The most straightforward method is explicitly defining dependencies for each service. However, when using this approach the information needs to be manually revised and updated. Another approach is intercepting the communication between parties. Finally, artificial intelligence techniques can be applied in order to find service dependencies.

Spotify's approach to services is based on domain names, which creates a new context for our research. The problem is that Spotify services do not describe the services on which they rely in a configuration file, currently this is purely implicit in the system owners' knowledge (i.e., in the implemented software and potentially in its documentation). However, given that all services run in the network are owned by one organization, full observation of all the traffic between them makes it possible to find these implicit dependencies.

Chapter 3

Method

This section gives an overview of possible methods for understanding service dependencies. Section 3.1 of this chapter addresses the problem of defining a service in Spotify's backend network. Assuming that hosts for the different services and the network links can be observed, then the observation of network traffic seems to be a promising way to learn about service dependencies. Therefore section 3.2 discusses possible methods for observing network traffic. Section 3.3 further investigates one of these alternatives, specifically collecting NetFlow information from the network equipment. Finally, section 3.4 gives a motivation and description of additional tools needed for the analysis.

3.1 Recognizing services

Whatever method of network flow monitoring is chosen, there should be way to associate an IP address, protocol, and port number tuple of a flow with a specific Spotify service.

In the original design, each Spotify service was expected to utilize specific non-overlapping ranges of TCP ports. Therefore it is tempting to classify services based upon these predefined static port number ranges.

However, after inspection of this information in currently operating network's DNS, it appears that there exist different services running on different machines, but using the same TCP port. One such an example is shown in the Table 3.1 [1]. A special script was written to analyze DNS records in order to find these *port clusters*. To conclude, both the destination IP address and destination port number are required to map network flow to the corresponding service based on DNS SRV entries.

Service	Min port	Max Port
_spotify-service1._http.spotify.net.	8081	8087
_spotify-service2._http.spotify.net.	8081	8087
_spotify-service3._http.spotify.net.	8082	8085
_spotify-service4._http.spotify.net.	8082	8097

Figure 3.1: *Example of overlapping use of TCP port numbers by different services*

As each host providing some specific service is usually tagged with a configuration management class, an alternative approach of associating a network flow with a service is based upon this class name. It is important to note that only the destination IP address is necessary for such a classification. In addition, the assumption that each host serves only one service must be valid.

3.2 Monitoring network traffic

One of the biggest implementation decisions to be made is how to monitor traffic. As we are interested in statistics at the transport layer, rather than at the application level, we simply need to collect information about traffic flows within backend network. Two basic approaches to finding traffic flows are logging information at each server machine or sniffing packets flowing through the network.

The advantage of sniffing traffic is that it is potentially more efficient and avoids use of resources on the individual server machines. However, the amount of traffic passing

[1]Note that the specific service names have been replaced by service(n) to protect proprietary details of the company's backend configuration. The details of which service is what is not important for our tools - only that there are dependencies, but because there is a many to one mapping between services and port numbers the DNS information is not sufficient to compute a unique inverse from port number to service.

through the routers of the internal network is on the order of terabytes per day, therefore some form of random sampling could be used to reduce the amount of traffic that has to be processed, but this occurs at the cost of introducing some bias into the results.

Tracking network flows on each of the server machines could offer greater flexibility. Moreover, this logging might be limited to only a short period of the day, in order to reduce the negative effects on each server's performance. Of course performing the logging in the servers themselves will also potentially introduce bias, which could range in magnitude from small to very large depending upon the other demands upon the individual server.

In the following sections we will examine several potential methods for tracking network flows. We have assumed that each of the servers are running a Debian based Linux operating system, as this was true at the time of writing this thesis.

3.2.1 Auditd tool

Auditd (Audit daemon) is a user space component for linux system auditing [9, 10]. It writes logs for system calls to disk. It is controlled by several tools, *initctl* for changing rules and *ausearch* and *aureport* for manipulating the logs.

This tool can track operations at the level of system calls. Discussion in [11] explains how to add rules to listen for system calls. We experimented with adding rules for one kind of system call: socket calls. The authors of [12] and [13] gives guidelines for analysing *auditd* logs. However, the information which is logged is difficult to parse as the format depends on the kernel version and is encoded in low level codes (figure 3.2 shows an example of part of an *auditd* log). In addition, while it is possible to understand when a socket is created, this does not provide information about the actual packets being sent over socket.

```
type=UNKNOWN[1325] msg=audit(1330001823.261:26): table=mangle family=2 entries=6
type=SYSCALL msg=audit(1330001823.261:26): arch=40000003 syscall=102 success=yes exit=0 a0=e a1=bffe35d0 a2=b7824ff4 a3=8143f20 items=0 ppid=1370 pid=1458
auid=4294967295 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=(none) ses=4294967295 comm=”iptables” exe=”/sbin/iptables-multi” key=(null)
type=UNKNOWN[1325] msg=audit(1330001823.265:27): table=filter family=2 entries=6
type=SYSCALL msg=audit(1330001823.265:27): arch=40000003 syscall=102 success=yes exit=0 a0=e a1=bfa8be10 a2=b7786ff4 a3=94db0f0 items=0 ppid=1370 pid=1463
auid=4294967295 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=(none) ses=4294967295 comm=”iptables” exe=”/sbin/iptables-multi” key=(null)
type=UNKNOWN[1325] msg=audit(1330001823.269:28): table=filter family=2 entries=7 type=SYSCALL msg=audit(1330001823.269:28): arch=40000003 syscall=102
success=yes exit=0 a0=e a1=bfd44780 a2=b7873ff4 a3=86a6270 items=0 ppid=1370 pid=1464 auid=4294967295 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0
tty=(none) ses=4294967295 comm=”iptables” exe=”/sbin/iptables-multi” key=(null)
```

Figure 3.2: *Sample part of an auditd log file. Params a1, a2 ... correspond to arguments passed to the system calls*

3.2.2 Iptables

Another option for tracking TCP connections is using iptables [14, 15]. Iptables are already utilized within the Spotify network, and each machine has its own set of firewall rules (configured using iptables).

Iptables have additional module for checking the state of a TCP connection. This module is called conntrack [16]. There are only four states available: NEW, ESTABLISHED, RELATED, and INVALID. Unfortunately, these states do not correspond to TCP states, as shown in figure 3.3. These states instead relate to the internal state of conntrack and do not accurately provide information about the TCP connection's state.

The conntrack state NEW simply means the packet is the first packet in a *conntrack* tracked connections (most probably, that will be a TCP SYN packet). The ESTABLISHED state means that *conntrack* has seen traffic in both directions. The RELATED state connections are the ones which are connected to another ESTABLISHED connection. Finally, INVALID connections do not belong to any category and should generally be dropped.

In the TCP three-way-handshake context there are three messages before a TCP connection is set up: SYN, SYN + ACK, and ACK. This is where a slight, but important difference between the *conntrack* states and the TCP socket's states is.

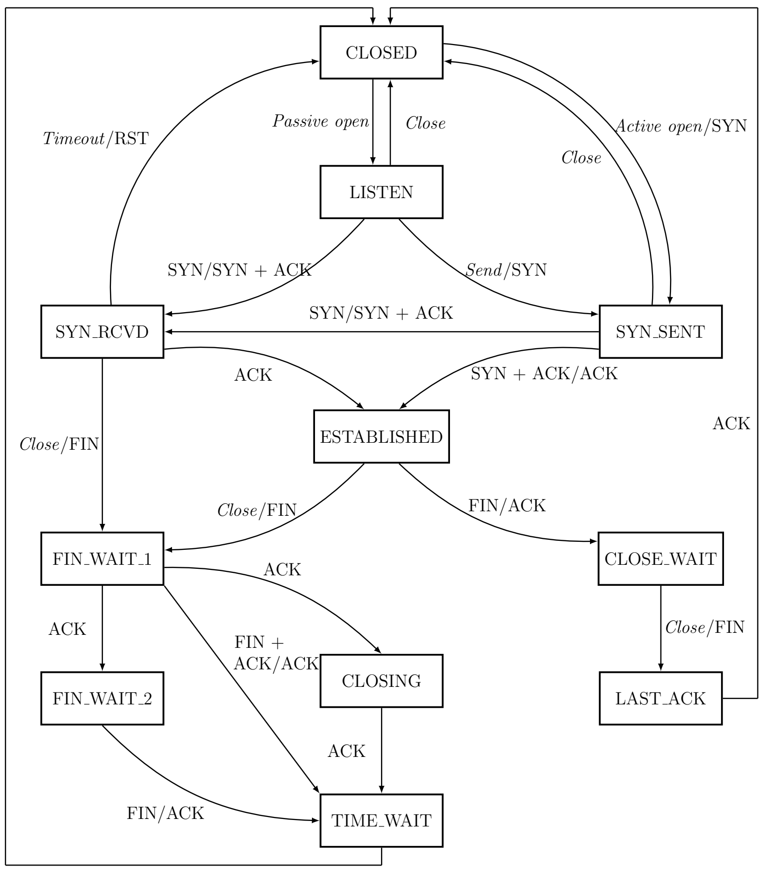

Figure 3.3: *TCP state transitions*

Therefore one way of tracking connections is logging a packet when the TCP connection is in the NEW state (first packet has been seen). To identify the end of the connection we can wait for a FIN from the server. However, both of these approaches are only estimations. As the NEW state only means that there has been a packet seen on one side, but it is not yet agreed to initiate a TCP connection. As for a closing connection, the FIN packet simply indicates that the server wants to close the TCP connection. If it did not receive a FIN from the client, it should wait for this FIN, and even after that there is still some time (on the order of minutes) before the socket is actually closed.

To estimate the traffic bandwidth for each connection, one could enable SEQ number logging in the iptables. Having the sequence number for the last packet and the first packet, it is possible to estimate the total number of bytes sent. The iptables manual [17] explains these additional logging options, including how to log sequence numbers.

We have experimented with these ideas, setting up the required *iptables* rules and testing whether it works in practice. However, despite the mechanism working, the biggest issue is its efficiency. In particular how much the performance of the server decreases after introducing these changes in the iptables. In addition, the logs will quickly become very large and this solution does not scale to collect data from large number of machines.

3.2.3 Netstat tool

Another solution is using system tool *ss* (or *netstat*) to list all opened sockets. Fortunately, this is not a process blocking operation, therefore it is efficient and causes minimal interference with the operation of the server. However, this approach needs polling to capture connection start and end events. As a result this approach is most useful when only a snapshot of connections is needed. Finally, this approach does not enable us to collect information about the number of packets or bytes transferred, but rather simply the number of connections.

3.2.4 Collecting NetFlow information from network equipment

A more passive monitoring approach is to collect the flow data exported by routers or other network equipment using the NetFlow protocol [18]. Fortunately, information about flows is sufficient for this master thesis project's goals. As a positive side-effect, this flow information can be used for other purposes within organization - for example, for analysis of Denial of Service (DoS) attacks. However, these additional uses of this data lie outside the scope of this thesis. As this approach seems the most efficient and suitable approach of collecting data, it will be more thoroughly discussed later in this thesis.

3.2.5 Exporting NetFlow using fprobe

Sometimes NetFlow data emission from the network equipment is not feasible or desired. An alternative solution is to use a NetFlow collection infrastructure which offers the advantage of good scaling, but this process is based upon emitting NetFlow data from the hosts rather than from the network equipment. *Fprobe*, a *pcap* based Unix tool, works in exactly this way.

3.2.6 Exporting NetFlow using a custom tool

NetFlow data can be emitted using the *fprobe* tool. Unfortunately, the *fprobe* tool is based on the *pcap* library and its operation is costly as it must actively listen on interface in promiscuous mode.

However, it is possible to exploit the fact that Spotify uses *iptables* on hosts to create a more efficient network information collection tool using the *libnetfilter_conntrack* library [19]. This is a user-space library providing an application programming interface (API) to the in-kernel connection tracking state table. Reusing auxiliary data about connection events from the iptables *conntrack* module is potentially more efficient than active listening on an interface.

To experiment with this idea, we created a *Python* script which registers listeners for NEW and DESTROY connection events to the library. We used Python *ctypes* module in order to use the functions from the dynamically loaded library (DLL) of *libnetfilter_conntrack*. Each event contains information about the network flow, in particular, source and destination IP addresses and port numbers, protocols of transport and network layers, and a conntrack connection ID. This Connection ID can be used to identify starting and ending events associated with the same connection. After receiving the connection DESTROY event, a NetFlow record for this connection is created. Several NetFlow records are then packed together[2] into one NetFlow protocol version 5 packet and transmitted via the UDP protocol towards an active NetFlow collector. We used the Python packet creation and parsing

[2]The NetFlow protocol allows up to 30 records in one packet.

library *python-dpkt*[3][20] for creating NetFlow packets[4]. Regular Unix sockets were used for transmitting these packets. This script must be configured to work as a daemon process. The drawback of this tool is that it may require installing additional packages (such as *libnetfilter_conntrack* and *python-dpkt* dependencies) and a restart of the machines.

[3]There are alternative packet craft tools, such as Scapy.

[4]The latest release needs a patch due to the existing bug in NetFlow packet length calculation.

3.3 NetFlow

The previous section gave an overview of possible traffic monitoring approaches. The option of emitting NetFlow data from network equipment has an advantage of low impact on the server's performance. This is the reason why this section analyses the use of the *NetFlow* protocol to collect information about network traffic flows[5]. We will use this information for backend service topology analysis.

First, the *NetFlow* protocol will be introduced. After this an overview of various NetFlow based monitoring and analysis suites will be given. We will motivate our choice of the *SiLK* suite and describe how it can be deployed on a large network, specifically one that interconnects several data centers. Such a network of multiple data centers, each with many machines, is typical of a backend network, such as one used by Spotify.

3.3.1 NetFlow protocol

NetFlow is a network protocol originally developed by *Cisco*. It has been become a standard and is available on other platforms, such as *Juniper OS* and *Linux*. The protocol provides aggregate information for each network flow and can be emitted by the routers and other network equipment. Flow collection is based on an aging counter: a received packet which corresponds to already monitored traffic flow resets this counter for the flow. The close of a *TCP* connection forces the device to export the flow related information immediately. Among other fields, a *NetFlow* record contains information about source and destination IP adresses, port numbers, protocol used, and number of complete bytes sent. Table 3.1 shows the complete format of all the fields contained in a *NetFlow* record. NetFlow packets are usually sent via the UDP protocol, because flow data is not critical and it may create a big traffic load.

Version v9 of *NetFlow* was extended to support IPv6, MPLS, and other new features [18].

[5]Note that NetFlow data emission and collection are two distinct processes. NetFlow data collection can be done using SiLK suite, while NetFlow data is emitted either from network equipment, or hosts (by using fprobe or fcollector tools)

Table 3.1: *NetFlow v5 record format [21]*

Bytes	Field	Description
0-3	srcaddr	Source IP address
4-7	dstaddr	Destination IP address
8-11	nexthop	IP address of next hop router
12-13	input	SNMP index of input interface
14-15	output	SNMP index of output interface
16-19	dPkts	Packets in the flow
20-23	dOctets	Total number of Layer 3 bytes in the packets of the flow
24-27	first	SysUptime at start of flow
28-31	last	SysUptime at the time the last packet of the flow was received
32-33	srcport	TCP/UDP source port number or equivalent
34-35	dstport	TCP/UDP destination port number or equivalent
36	pad1	Unused (zero) bytes
37	tcp_flags	Cumulative OR of TCP flags
38	prot	IP protocol type (for example, TCP = 6; UDP = 17)
39	tos	IP type of service (ToS)
40-41	src_as	Autonomous system number of the source, either origin or peer
42-43	dst_as	Autonomous system number of the destination, either origin or peer
44	src_mask	Source address prefix mask bits
45	dst_mask	Destination address prefix mask bits
46-47	pad2	Unused (zero) bytes

However, the prevailing version of the NetFlow protocol is version 5, therefore this version was chosen for my imlementations. Another variant of NetFlow is Sampled Netflow (*sflow* [22]). In *sflow* random sampling is introduced in order to make data collection more efficient. In this case only a subset of packets are monitored, therefore flow records contain estimations, rather than real traffic information. Such sampling may induce errors in analysis, as discussed in [23].

3.3.2 NetFlow based tools

There are several free NetFlow based tool suites available. This section gives an overview of the most widely used tools.

Flowd [24] is small, fast, and secure *NetFlow* collector. This tool focuses on storing

NetFlow data on a disk. As an advantage of this tool is that it offers *Python* and *Perl* interfaces for raw data access. However, the lack of analysis tools is a drawback.

Fprobe [25] is a probe that collects network traffic data and emits it as NetFlow flows to a specified collector. Although it is difficult to use as a monitoring tool alone, it is a vital tool for testing, as will be described in the following sections. IP traffic from a local machine can be transformed into *NetFlow* data, enabling testing even before routers are configured to export *NetFlow* data.

Nfdump [26] collects and processes data according to a command line interface. Nfdump is a part of the *Nfsen* project. Although it has scalable data collecting features, it lacks analysis tools. In contrast, *Nfsen* is a related project which can be used to visualize the NetFlow data via a web interface.

Flowtools [27] is library and a collection of programs used to collect, send, process, and generate reports from NetFlow data. Flowtools was developed by Mark Fullmer. The architecture of this suite and tools is similar to the *SiLK* suite, described in the following paragraph. A practical disadvantage of the suite is that it is supported by only one developer and lacks good installation and usage documentation.

System for Internet-Level Knowledge (*SiLK*)[28] is a NetFlow collection and analysis suite for large networks. It is designed to scale up to support very large networks. It is possible to merge data from different collectors and send incremental data files across the network. In addition, SiLK has a powerful analysis suite which enables fast filtering, sorting, grouping, and other analysis operations. Therefore, *SiLK* is a good candidate as a tool for Spotify's backend service correlation analysis. This suite has both sufficient collection and analysis tools. The major reason why it is proposed as the most suitable suite is that it provides a scalable and secure data collection solution. The following section will give further details about SiLK.

3.3.3 SiLK suite for NetFlow

SiLK is a analysis and traffic collection suite developed and maintained by the U.S. CERT Network Situational Awareness Team to facilitate security analysis of large networks. It enables effective collection, storage, and analysis of network data. It runs on most UNIX style platforms. The SiLK suite is a collection of tools, which must be configured to work together to achieve the desired effect. The most important tools are:

rwflowpack is a core tool which packs NetFlow data. It listens for the network devices which produce NetFlow records, converts these records to SiLK format[6], and then packs and stores this data on disk for later processing or transmission to other network nodes.

flowcap simply listens for NetFlow data and prepares it for transmission. It is usually used when a collector must be near the NetFlow emitter, but when the packing of data may be done on separate machine.

rwsender sends files to a receiver. It watches an incoming directory for new files and sends them to a receiver when new files are observed.

rwreceiver creates a connection with *rwsender* and fetches incoming files, which are stored to an output directory. Other tools, such as *rwflowpack* or *rwflowappend* usually observe this directory and automatically process files which are placed there.

rwflowappend receives small batches of NetFlow records and stores them in tree organized by hour.

3.3.4 SiLK Deployment scenarios

The best layout for a SiLK monitoring infrastructure depends on various factors, including analysed network size, number of collectors, hosts' computing power, and others factors. Some of possible usage scenarios are depicted in the SiLK installation guide [29].

[6]Properiatary storage on disk format

In the simplest single machine scenario, a dedicated machine listens for NetFlow data and stores this data on its disk. Analysis of this data is also done on the same machine. In this case it is sufficient to run the *rwflowpack* tool. This approach has the advantage of being a very simple configuration (as shown in figure 3.4).

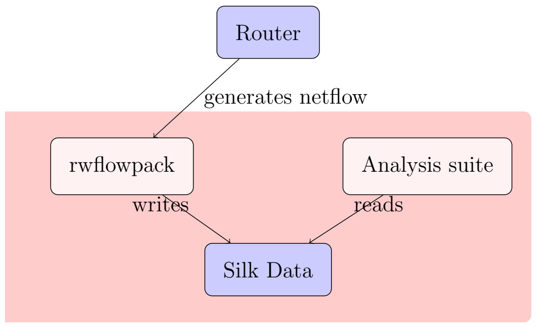

Figure 3.4: *Single machine scenario. Flow packing, storage and analysis on the same machine*

In the second scenario (shown in figure 3.5), data is collected locally and stored remotely. In this scenario collectors also pack data by running *rwflowpack* and using *rwsender* to transmit data to the remote analysis machine. At the analysis machine this data is stored and processed after being automatically fetched the by *rwreceiver* tool.

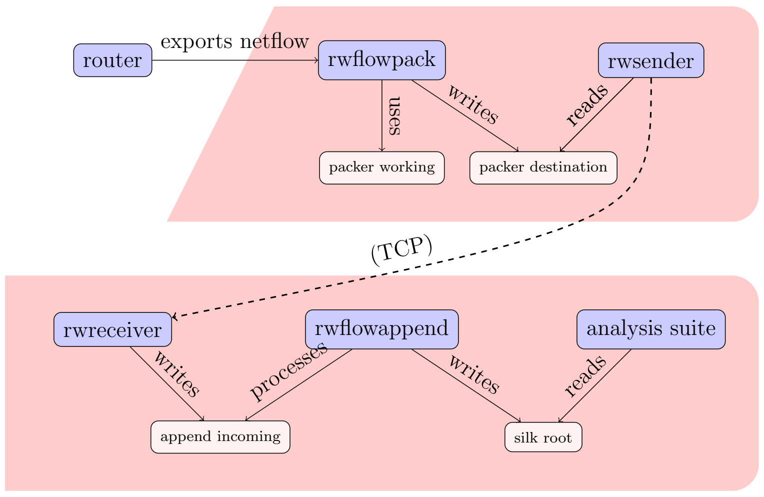

Figure 3.5: *Local collection and remote storage scenario.*

In the most complex scenario (shown in figure 3.6), data is collected remotely by collectors, then transmitted and packed on dedicated remote machines. Therefore these packing machines neither collect, nor store flow information for analysis. They simply forward the collected data. The difference between the scenario depicted in figure 3.5 and this more complex scenario, is that it may utilize several analysis machines and the NetFlow data is packed and analysed at different locations. This more complex scenario facilitates scaling up the amount of NetFlow data that can be analyzed and enables this analysis to present the analysis closer in time to when the data was collected.

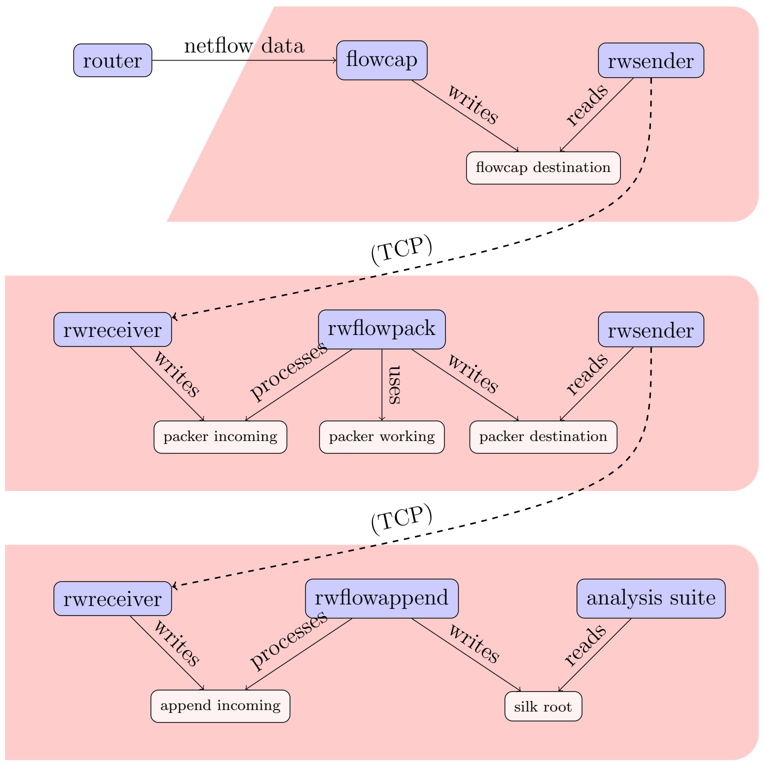

Figure 3.6: *Remote collection and remote storage scenario*

The most suitable scenario to be deployed in the case of Spotify is to use the intermediate scenario depicted in figure 3.5 with local collection and remote storage. Given that separate powerful machines may be allocated for monitoring, it would not be efficient to run only collectors on the monitoring machines (as was shown in figure 3.6). Instead collectors can also pack data before transferring it to other machines for analysis. Therefore a single central analysis machine could organise and append packed data coming from different collectors. Scalability is also greater when main computations and data processing is moved towards collectors (i.e., exploiting distributed data processing to reduce the amount of data that has to be distributed and analyzed). The simplest scenario (as was shown in figure 3.4) may be not suitable due to big load on one processing machine and the fact that collector can be too distant from NetFlow sources.

3.3.5 Deployment of SiLK

To facilitate deployment of the SiLK suite, we had to configure SiLK to work in the production environment. We had to adjust the package to work with the automated configuration management software used in Spotify. This configuration was done in such way that it would be easy to add extra NetFlow collectors. The connections between collectors and analysis machine were secured by using TLS (a built-in feature of SILK *rwsender* and *rwreceiver* tools).

3.3.6 Configuring exporters

Juniper Networks knowledge base [30] contains an example of setting up NetFlow data export in JunOS. In order to export NetFlow data, a sampling firewall rule should be defined. A sample configuration of such firewall rules is shown in appendix A.

3.4 Creating analysis tools

Information about network flows must be processed in order to find and analyse the backend service dependencies. Although SiLK provides some command line tools for analysis, its focus is primarily on information retrieval, rather than complex processing.

This section introduces the SiLK based analysis framework which is essential for our NetFlow data analysis. First, an overview of the available SiLK analysis tools is given. After pointing out the limitations of these tools with respect to the thesis project's goal, the motivation and specifications for a custom analysis tool are given. This is followed by a description of the design an implementation of such tool. This tool will be used as a base for the analysis of the NetFlow data presented in the next chapter.

3.4.1 SiLK as an analysis suite

SiLK contains various command line analysis tools. Some of these tools are:

rwfilter	for filtering packets by time, IP addresses, or port numbers
rwcut	for displaying fields of filtered packets
rwuniq	for grouping packets based on the value in a specific field

Although for some specific set of analysis tasks these tools may be more than sufficient, there are some important considerations which must be considered as we plan our data analysis.

Command line tools In network monitoring it is important to understand data, hence being able to display this data visually is advantageous. However, SiLK only provides command line based analysis. Connecting these tools to well known plotting suites such as *GnuPLOT* requires manual scripting.

Limited filter and grouping tools is limited by the SiLK tools, for example, it is practically impossible to use a combination of filters (such as the logical operators *OR* and *AND*). Similarly, only one level of aggregation is allowed with a grouping tool.

Limited granularity Because SiLK data is stored on the disk and organized in hourly buckets, in some situations the tools are limited to aggregating data only by the hour. When investigating special cases, such as a *DDoS* attack which could occur in a short time period, there is a need for greater granularity.

Incomprehensive usage In order to master SiLK analysis tools, it is important to understand the strict format for each command and its options. For example, when providing a time bin interval in the grouping tools, the parameter is passed as a number of seconds. This may be inconvenient when time bin is in order of hours. Some tasks also require using extra *Unix* command line tools, such as *awk*, *sort*, and others - making it even more confusing for the user.

3.4.2 Motivation for a specialized analysis tool

In order to understand the dependencies between services, it would be useful to present visual plots of the data. However, given the tools available with SiLK, it was clear that some additional tool was necessary for converting SiLK data about netflows into meaningful plots. One of the reasons for this is that the definition of a service in the Spotify's context can not easily be directly expressed in terms of netflows. Furthermore, grouping by service is not possible using the built-in SiLK analysis tools.

As with all analysis tools, even when there exist predefined visual plots, great value is added if the presentation of data is suitably adapted to the user's need. For example, one could be interested in the correlation of subsets of services, or services deployed only at one Spotify *site*.

The necessity of advanced data processing and plotting led to a solution which grew into a small framework for analysis. A benefit of the proposed framework is that this framework can be used for solving problems other than simply finding and monitoring service dependencies.

3.4.3 NetFlow analysis and plotting language

In order to ease the process of NetFlow information analysis, we devised a small language. This language is as simple as possible, allowing grouping, filtering, and importing custom functions from *Python* modules.

There have previously been attempts to create query languages for the NetFlow data in order to improve the analysis process. In one of these attempts, V. Marinov defined a query language in order to describe and identify the occurrence of network transform patterns in a collection of flow records [31]. The main purpose of this language is to detect various attacks. Several other tools use SQL based syntax. The final group are tools which use Berkeley Packet Filter syntax (such as one used in tcpdump tool). However, as far as we know, SiLK has the most advanced tools for analysis and grouping, although as we have seen it is still incomplete with respect to goal we wish to achieve.

The grammar of the language that we propose can be described as follows in BNF:

```
<program>    ::= <statement> ; <statement> ..
<statement>  ::= filter <expression>
              |  gfilter <expression>
              |  group <expression> as <name>
              |  emit [<expression>, ..] (expression, ..)
              |  import <literal>
              |  option <literal> <literal>
<expression> ::= <expression> + <expression>
              |  <expression> * <expression>
              |  <expression> / <expression>
              |  <expression> or <expression>
              |  <expression> and <expression>
```

```
                |  (<expression>)
                |  <name>(<expression>,..)
                |  <literal>
```

In the following description of the proposed language, information about a NetFlow record is referred to simply as a *node*. This is the smallest data unit for data processing. A set of nodes form a *context*.

Statements are executed sequentially in their order of presence in the script. With each statement the interpreter goes deeper in its recursion level until it reaches an *emit* statement.

The grouping statement *group* uses a function to transform a *context* into potentially several *contexts*. The grouping function accepts a NetFlow *node* as an argument and returns a key. The meaning of the key depends on the grouping function. For example, it can be the configuration class of the host identified by the destination IP address, or a timebin index identified by a starting time. *Nodes* which evaluate to the same key are grouped together and form distinct contexts. The following statement will be executed iteratively in each of the newly created contexts.

The filter statement *filter* uses a function to filter a set of *nodes* from a *context*. The filter function accepts a *node* as an argument and returns a positive integer value if this *node* passes the test. The filter is iteratively applied to all nodes in a context. Nodes which are accepted by filter form a new context, which is passed to the following statement.

The emit statement *emit* is used to output some data which can be plotted. As an optional argument (specified in brackets) it accepts a list of expressions, which define a key. If a key is not specified, then all keys from the previously executed *group* statements are used as a key. A required argument is a list of *expressions*, which define the emitted values. Therefore an *emit* statement emits a tuple of keys and values (for example, a key could be weekday, and value could be number of connections).

An expression evaluates to an integer value. Logical (Boolean) values are simulated as integers, with positive values meaning the boolean value *True*. Expressions may contain regular arithmetic operators (such as addition and multiplication), logic operators (such as *and* & *or*), parenthesis, and function calls. Either built-in, or custom functions can be called. The latter functions are imported using the *import* statement. Among the most important built-in functions are *count* and *sum*. The *count* function returns the number of nodes in a context. Since a and as node is a NetFlow item, the value of *count* literally means the number of connections[7] for the current context. Similarly, the *sum* functions iterates through all nodes in a context, fetching the attribute specified as the parameter for the *sum* function, converts the value of this attribute to an integer and calculates a total sum. For example, *sum('bytes')* would count total number of bytes for all NetFlow nodes in a *context*.

Section 3.4.5 gives examples to demonstrate language features, syntax constructs, and potential uses. Appendix C gives a more detail specification of available language constructs. Appendix D gives an overview of available built-in functions.

3.4.4 Extendibility of the language

There is also an *import* statement which allows the programmer to import custom *Python* groupers or filters to extend the framework. This flexibility is essential for our thesis project's goal, as we can define grouping of NetFlow information by service for the Spotify's context.

The importing module of the interpreter recognizes groupers by classname. Custom groupers should extend the class *BaseGrouper*. All functions in the custom module are classified as filters. We chose to implement groupers as classes, as they iterate through many nodes and may need initialization or precalculation to speed up a grouping process. Our interpreter is case insensitive and does it best to identify the function if it is not fully qualified.

[7]Since usually exactly one NetFlow record is generated for each connection, these terms are used interchangeably. Flow is exported as soon as the session is closed, which is identified by TCP-FIN (finish) or TCP-RST (reset) packets.

3.4.5 Examples of scripts

This section gives several examples of how the proposed language can be used.

The script shown in figure 3.7 will help us to understand whether there was an increase in the number of different source IP addresses sending IMCP pings during a suspected *DDoS* attack. The first line filters out the ICMP records. The second line divides the data into time bins of one minute duration. Finally, *diff* in the third line calculates number of different source IP addresses ("sip") in each timebin.

```
filter protocol("icmp");
  group time(bin="1 minute");
    emit diff('sip')
```

Figure 3.7: *Looking for ICMP pings during a DDoS attack*

Increasing number of different source IP addresses may indicate having a DDoS attack. The program shown in figure 3.8 can be used to further investigate the country of origin causing an overload. The first line groups a defined time range into timebins (with each timebin's duration defaulting to a one minute period). The second line divides the data in each timebin into context source country, which is determined by the source IP address. Lastly, *sum* in the third line calculates the sum of bytes for each context, defined by the nested grouping.

```
group time(from="yesterday", to="now");
  group country("sip");
    emit sum("bytes")
```

Figure 3.8: *Which country's requests generated the most traffic in the last 24 hours*

The last example shown in figure 3.9 produces the service dependency graph as a stacked bar chart. In this case *dip* indicates the destination IP address, while *sip* indicates the source IP address, hence the correlation of services with a given value of *sip* and given value of *dip* is a dependency.

```
group servicegrouper('sip') as service1;
  group servicegrouper('dip') as service2:
    emit [service1, service2] sum("bytes")
```

Figure 3.9: *Correlation of flows from various services*

3.4.6 Implementation of analysis framework

The helper framework is written in the *Python* programming language and is based on a top-down parser. This parser uses the *Python* lexer to convert a program's source code into lexems and a custom tokenizer to convert these lexems into tokens[8].

A third party *Python* library is used for the actual plotting. Among the several candidate plotting libraries, we chose to use *MatPlotLib*[32], because it is stable, well supported, and has a very good gallery of examples.

Our framework consists of the following components:

Lexer	Decorates Python lexer to split script source into lexems
Tokenizer	Uses top-down approach parsing to create a syntax tree of tokens
Runner	Runs statements from syntax tree
Grouper	Runs grouping logic
Importer	Run-time python module loader
Builtin	Contains some built-in functions such as geolocation groupers; functions *sum*, *diff*, and *count*
Context	Data structure for NetFlow data storage. A context consists of *nodes* (NetFlow records)
Flusher	Flushes program output to plotter

[8]It was also possible to use an open source based parsing libraries to build token tree, but the custom parser is easy to extend and was suitable for this thesis project.

Plotter	Makes the best effort to render given output data as a plot

Figure 3.10 depicts the overall design of the system.

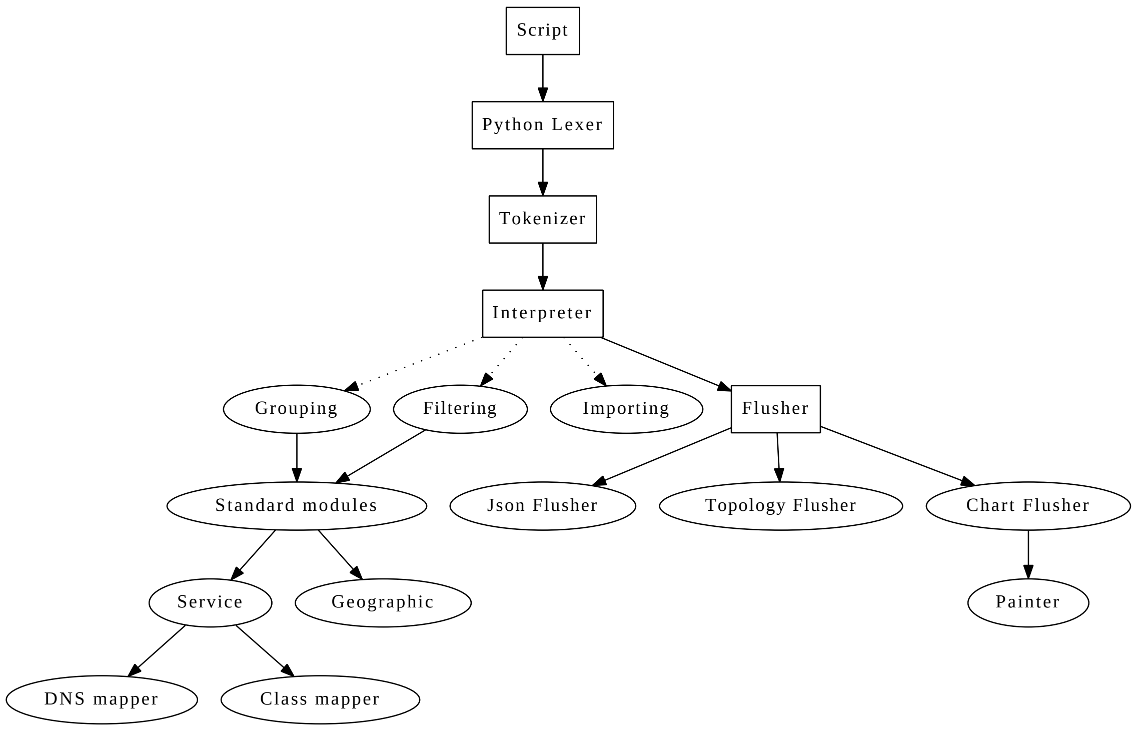

Figure 3.10: *The design of the analysis framework*

Some attributes of a NetFlow node can be derived from other existing attributes, for example a source country code can be derived from the source IP address. These derived attributes are static and do not change during runtime. In addition, they might even be not needed in calculations. Therefore these derived attributes support a lazy loading and caching mechanism, thus they are only evaluated when they are actually used and this is done at most once. Lazy loading is done transparently to other classes.

3.4.7 Implementing custom groupers

Geo grouping. Grouping by geographic location (according to the source or destination IP address) can be useful for various purposes. For example, it can be used to examine where the customers of a specific Spotify access point are based.

Grouping by the geographic location was added to our framework by using the *Python GeoIP* library [33]. This library provides functions to fetch a country code or the full name of the country given an IP address. Our geographic grouper takes one argument as an additional parameter to indicate what IP address should be used for this translation (i.e., the source or destination IP address) and utilizes functions from the *GeoIP* library in order to return a country code as a string.

Our geographic filters exploit the same library to filter NetFlow nodes matching certain geographic criteria. Logic operators AND or OR should be used to support more complex filtering scenarios, such as filtering nodes for a group of countries.

Grouping by service. Despite rapid changes in network infrastructure, there are currently a limited number of ways to determine what is the destination service according to the NetFlow node. The most important attributes of a NetFlow record (i.e., node) are the destination IP address and destination port number.

As described earlier in section 3.1, one of the approaches is to use DNS information to recognize the service. The DNS is a binding glue in the backend system and is widely used to locate other services. The problem is that the complete picture of IP address to service mapping is necessary for analysis and grouping purposes. As it would be too expensive to query the DNS server for each service, instead we chose to parse DNS records instead. We used the *Python DNS* library[34] to transfer all DNS zone files. This Python library supports DNS security, which is needed to pass a secret key together with a query (as a complete zone transfer requires authorization). Following the transfers, we extracted all SRV and A type records. As DNS records may be changing rapidly in the DNS servers, there exists a potential problem of DNS records being already invalid during the analysis process. This is further discussed in the Future work section. However, in the case of real time analysis there is only a small chance that the DNS records will be altered between the time of lookup by the client and the time of the lookup by the analysis tool. This is futher discussed in section 5.2.

The SRV records map service name and protocol tuple into list of host names and port numbers which define service providers. The host names can be resolved to IP addresses. Using this information we created a reverse entry for each SRV record, this way generating the required mapping that will be used later to identify the flows. It is guaranteed that each host and port number pair will have an unique SRV record associated with it[9]. However, this mapping must be updated regularly (for example, several times per day), because DNS records are altered regularly.

Another approach is grouping NetFlow records by configuration management classes associated with a destination host (further referred to simply as *classes*). However, these classes do not always correspond to services. In addition, some hosts may be tagged with several classes. In the current implementation, when there are several classes associated with a host, then the first one in alphabetical order is picked. However, such a situation is not usual. An alternative approach in such cases would be to use another type of ordering, for example by priority associated with each class. Such a priority list must be managed and that is a major disadvantage of a class based approach. Finally, some classes may be irrelevant for the analysis of services. Despite these disadvantages and need the for manual supervision, grouping by host class can be a potentially successful way to identify a service.

Caching. The information about *DNS* records and configuration management classes corresponding to the hosts may change rapidly. Therefore it is desirable to apply caching mechanisms. Our framework supports caching of data of an arbitrary time interval (which defaults to one hour), so that processing consecutive queries takes less time. Cached data is stored in *JSON*[10] format files on the disk. Service mapping information is fetched from these files, unless the file modification date has expired. In such case the cache is updated to reflect the recent situation of DNS and host classes.

3.4.8 Connecting analysis tool to SiLK

Our analysis supports two methods of inputting NetFlow data: JSON and standard input.

[9]Generally such a mapping might not be unique, however its uniqueness is ensured by the Spotify's architecture.

[10]Object serialization format

The JSON input method reads NetFlow data from a JSON object which has a strictly defined structure. This approach is useful if suites other than SiLK are used for NetFlow collection and storage. In addition, it enables offline processing and data migration.

The default input method is reading from standard input. The *rwfilter* and *rwcut* tools of the *SiLK* package are used to print network flows to standard output. The resulting standard output is then redirected to the analysis framework by using Unix pipes.

Chapter 4

Data Analysis

The goal of this section is to analyze backend service correlations using information about netflows. These correlations are best depicted using the tool we developed for this purpose.

Section 4.1 provides an analysis of one of Spotify's services using our framework. It describes what is the expected behavior of this service and compares it with the results obtained from network data. Section 4.2 describes how a full Spotify service topology tree can be obtained by using these tools. Two possible approaches of classifying network flow as services are compared in 4.4. Section 4.5 compares the efficiency of two possible methods of collecting network data from the hosts.

4.1 Analysis of sample Spotify service

This chapter provides an analysis of one of Spotify's backend services - browse (further referenced to simply as *browse*) using the collected NetFlow information and the analysis tools built for this purpose. The *Browse* service looks up metadata data and acts as a mediator which combines data into browsable results. It returns XML or JSON based upon a request from other services.

According to services documentation, this service should forward requests to the following services:

Search All search requests pass through browse. The requests are forwarded to search which responds with lists of IDs.

Toplist Works similarly to the *search* service, but returns a truncated list of matches.

The communication between these services and the service *browse* have an expected behaviour. It is important to note that communications patterns not necessarily correlate to actual Spotify users' behaviour. Sample data was collected for a one hour interval from one of the production machines which runs the *browse* service. The analysis in the following sections is based on this data. Scripts which perform an analysis and generate plots were written in our new special purpose language (this language was described in section 3.4).

4.1.1 DNS based service classification

One of the possible ways to classify outgoing connections is based on DNS SRV records as described earlier in section 3.1. As each service in Spotify has SRV records pointing to a specific location, defined as a hostname (translatable to an IP address) and port number tuple, it is possible to match each flow to a service based upon its destination IP address and destination port. Destinations that do not match any service are classified as *Other* (mostly this is communication with access points). The script shown in figure 4.1 performs a DNS based classification of destination IP addresses.

```
import 'services';
 group servicegrouper('dip');
   emit (count() means 'number of connections')
```

Figure 4.1: *Script for classifying outgoing connections using DNS*

This script produces a DNS based outgoing service classification chart. An example of such chart is shown in figure 4.2. It can be seen that the service *browse* actually communicates to services *search* and *toplist*, as was expected. In addition, this chart reveals communication to other services: *dist*, *search suggest*, *webservice*, and *info*.

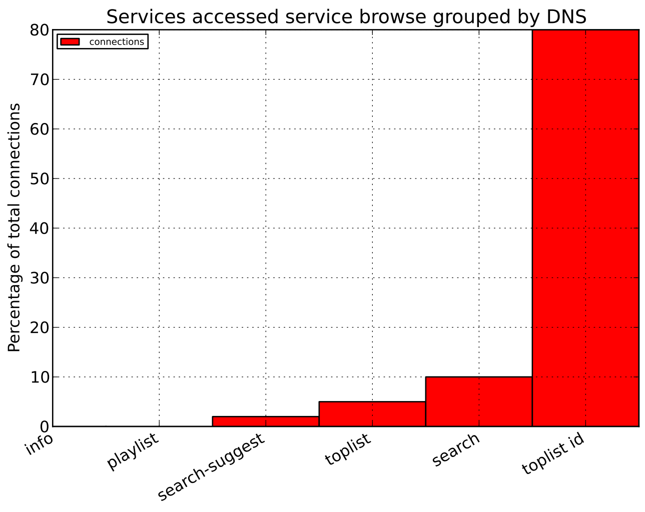

Figure 4.2: *DNS based classification of outgoing connections into services*

4.1.2 Configuration class based service classification

As each server machine usually is dedicated to one specific service, all of them are tagged with a specific configuration class name. This approach helps to scale new services and eases the migration process. Machines which run different Spotify backend services are tagged with different class names. Therefore fetching the class names associated with a host is also a good approach to classify services. The script shown in figure 4.3 performs a service classification based on configuration management classes.

```
import 'services';
 group classgrouper();
   emit (count() means 'number of connections')
```

Figure 4.3: *Script for classifying outgoing connections using host classes*

This script produces a configuration management class based service classification chart, such as shown in figure 4.4. In addition, connections were also grouped by geographic site. Because IP addresses do not reflect the geographic location of machines in Spotify's case, site classification is instead based the destination IP address. The chart proves that (aside from a small insignificant exception) the *browse* service communicates other services running on machines within the same *site* and therefore no inefficiencies were discovered.

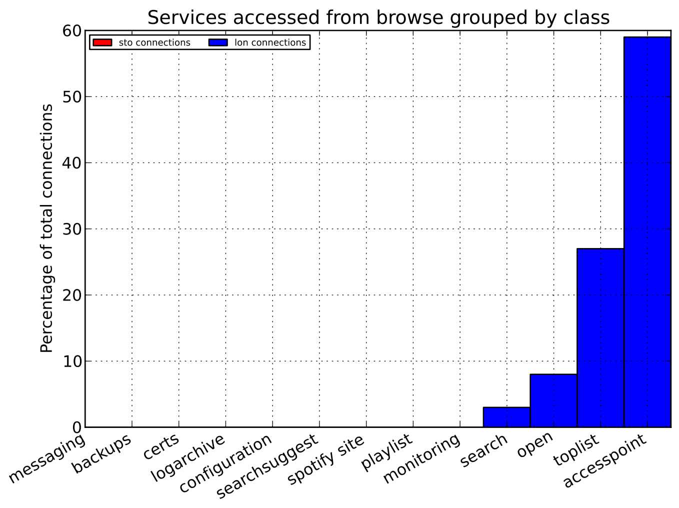

Figure 4.4: *Host-class based classification of outgoing connections into services.*

Let us introduce a few modifications to the chart 4.4. First of all, we indicate to sum the number of bytes transferred instead of the number of connections. In addition, we do not show access points, as we think that they should not be regarded as service. Finally, we skip services which are insignificant (comparing average bytes per second to a threshold).

The desired effect can be obtained by using the script shown in figure 4.5

```
group classgrouper('dip') as g1;
  gfilter not var('g1', 'accesspoint');
    gfilter bps() > 100;
      emit [g1] (bytes() means 'bytes')
```

Figure 4.5: *Script for advanced network flow filtering*

This script creates a chart, such as shown in figure 4.6. Note how the volumes for each service differ when number of bytes are measured in contrast to number of connections as it was shown in figure 4.4.

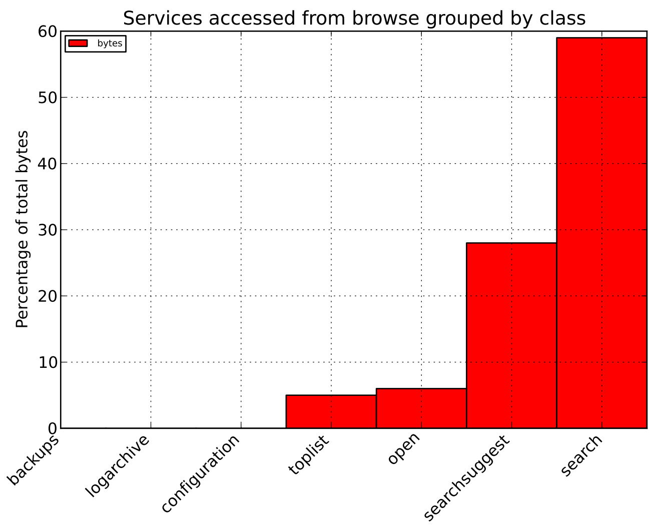

Figure 4.6: *Number of bytes transferred to various services based on host class based classification.*

Finally, figure 4.7 shows a classification by class additionally grouped by the network to which a destination IP address belongs. It can be seen that with exception of access points (which have public IP network addresses), all services communicate on interfaces within internal service network.

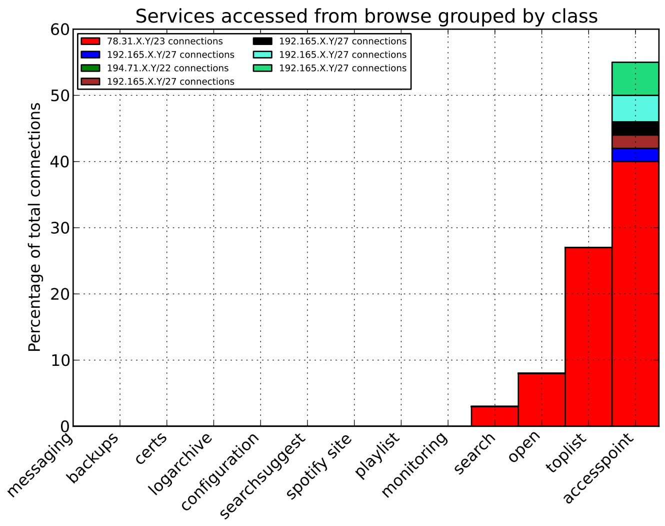

Figure 4.7: *Outgoing connections to various services (based on host-class classification) grouped by the destination network*

4.1.3 Overtime classification

It is also valuable to understand the dynamics of service dependency, i.e. how communication patterns and traffic volumes change within time. The script shown in figure 4.8 groups a one hour inspection interval into one hundred smaller timebins and gives a more detail view of services which are communicated in each of these smaller time intervals.

```
import 'services';
 group classgrouper() as g2;
  group time(bins='100') as g1;
    emit [g1, g2] (count())
```

Figure 4.8: *Script showing service communication dynamics*

The result of this script is shown in figure 4.9. This shows that communication from *browse* to most of services is smooth and not spiky with an exception of communication to an accesspoint. In this concrete example, communication to access point had around 5 minute duration repeating patterns.

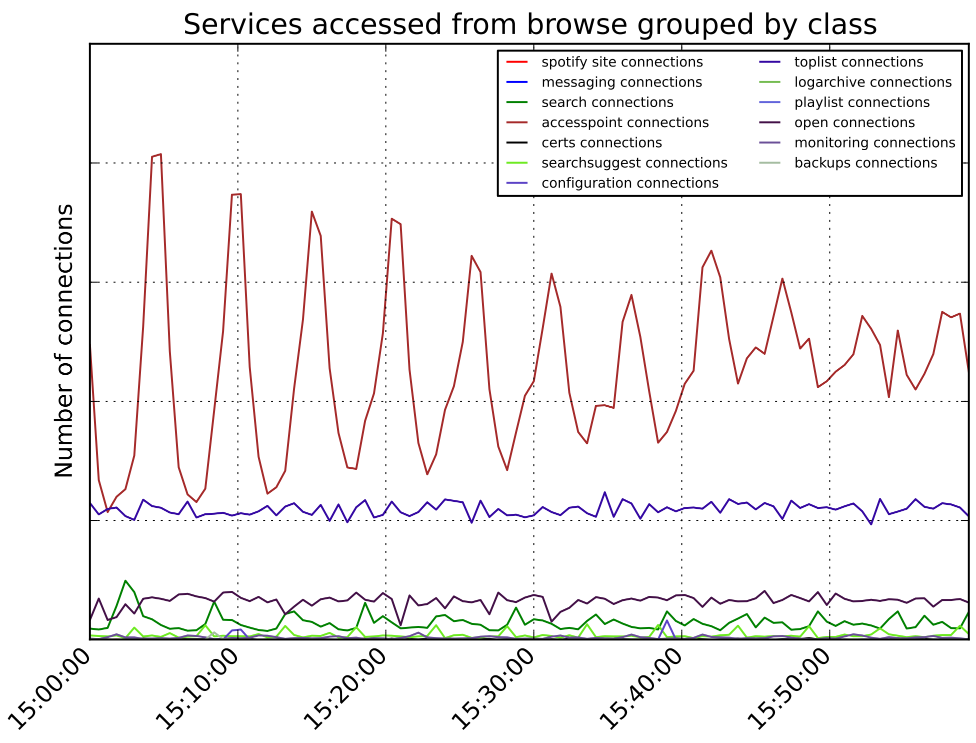

Figure 4.9: *Dynamics of the number of connections to other services (host-class based classification)*

4.2 Service Dependency Graph

Once network flows are classified into services, it is possible to construct a graph, depicting service dependencies. The detail semantic analysis[1] is not in scope of this thesis. The width (thickness) of the graph edge is determined using a logarithmic rule taking the number of connections as a parameter. The graph structure is then expressed in *DOT* language [35] and *Graphviz* tool [36] is used to create a visualization. An example of such graphic is shown in figure 4.10 [2].

The graph structure is created by a special module in our framework. The framework has a special flusher[3] called *GraphFlusher* which instead of rendering a chart, updates edges of the file storing the graphal structure. After creating a raw graph structure, postprocessing takes place. During this postprocessing the most insignificant edges (i.e., those not exceeding a configurable threshold) are deleted and the edge line width is calculated.

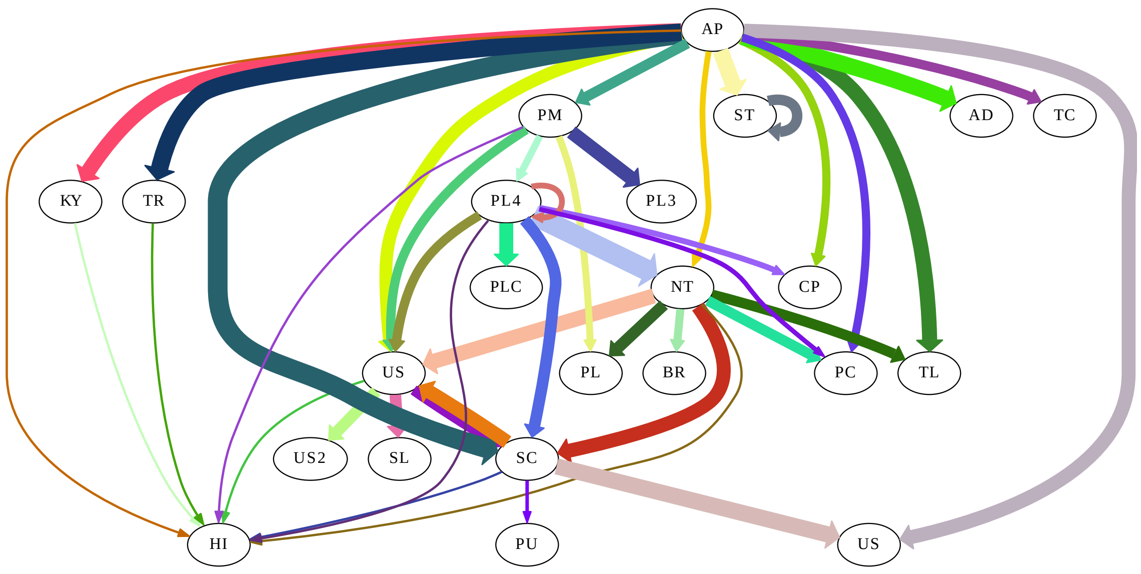

Figure 4.10: *Service Dependency Graph. Bubbles represent distinct Spotify service and edges dependencies between them.*

[1] For example, node's link to itself may indicate existing distributed service.
[2] The actual service names are hidden to avoid revealing company proprietary information.
[3] Section 3.4 describes flusher concept.

4.3 Expected vs. real service behaviour

The real-time data indicated a necessary update to an existing service topology graph which was obtained by interviewing system owners. However, the details of this process cannot be revealed due to the high sensitivity of this specific data. It is possible to automatically construct a graph which shows the differences between the real and expected topology graphs. However this is left for future work.

4.4 Comparing classification methods

We examined two way of classifying services. The first method is based on DNS and the second is based on configuration management classes. In this section we propose several performance evaluation criteria and use them to compare both methods. We define *results' accuracy*, *setup time*, and *execution time* as the three most important performance factors.

We define *accuracy* as the ability to classify networks flows to *services* as defined in this project as correctly as possible. The reference evaluation is based on expert knowledge. The *setup* time is defined as the time needed to perform precalculations when the cache is expired. The *classification* time is defined as time needed to classify network traffic information from a one hour period, given the cache (containing precalculated auxiliary data) is already available.

DNS based classification produced better *accuracy*. Although class based classification identified more network flows, it had an undesirable effect of including supporting services such as monitoring. Approximately 30% of classes does not directly relate to services. Fortunately, these classes usually constitute a small fraction of total network traffic volume and can be removed by postprocessing as was described in section 4.2.

The class based classification had a slightly smaller setup time, as fetching class information from database was on average slower then executing several DNS zone transfers (32.2s compared to 18.2s). Therefore initialization of the DNS based classification is faster.

Despite quite significant performance differences for precalculation step, the calculation of the DNS and class information caches is strongly dependent on the implementation and its performance could be optimized.

When auxiliary data was available, the actual classification time was similar for both methods (DNS based grouper is about 5% slower in classifying the same amount of flows).

In principle, it is possible to mix DNS and class based approaches into a hybrid classifier. However, combining information from two distinct sources in a meaningful way is not trivial operation.

Table 4.1 summarizes the above observations from these comparisons.

Table 4.1: *The comparison of classification methods*

Feature	DNS based	Configuration Class based
Accuracy	Higher	Lower
Setup	Faster	Slower
Classification	Similar	Similar

4.5 Comparing data collection methods

This sections compares two possible data collection methods. At the time of writing this thesis probably the most efficient method, NetFlow emission by the network equipment was still technical impossible[4]. Therefore as an transitional solution, we decided to collect network information from the hosts.

This was achieved by gaining access to a set of production machines for each key (critical for Spotify's backend and customers) service. We ran two different tools which aggregated traffic passing over the local interfaces, converted this data into NetFlow format packets, and transmitted these packets to a remote *SiLK* collector.

The first tool we have tested is the *pcap* based tool *fprobe* (described in section 3.2.5). This tool actively listens to the interfaces in promiscuous mode. Unfortunately, as all packets must pass through the central processing unit, this is potentially a costly process. Therefore we created an alternative data collection tool which exported connections from the iptables *conntrack* module (as described in section 3.2.6).

In order to compare these two data collection methods, we ran both tools seperately in two consecutive hours and measured their CPU usage and impact on the overall server performance. Table 4.2 shows measurements of the performance impact on these set of sample machines during the test[5]. The percentage of CPU resources used was obtained by dividing the CPU usage of the collector process by the total CPU usage[6]. The column *connections* shows the relative traffic volume among measured services in terms of the number of connections. The extended version of the resources of these measurements is available in appendix B.

[4]It seems that switches need a firmware upgrade before NetFlow can be enabled.

[5]The actual service names are obfuscated.

[6]Total CPU percentage was computed by running the Unix *ps* command and using *awk* to sum percentages. Note that total CPU may exceed 100% when machine has multiple cores.

Table 4.2: *CPU for data collection compared to total CPU*

Service	CPU for fprobe	CPU for fcollector	Connections
AP	3.7%	0.8%	12.1%
SC	4.7%	0.0%	26.2%
US	4.5%	2.4%	12.0%
PM	5.7%	1.0%	5.0%
PM4	7.7%	0.1%	1.3%
KY	4.1%	1.4%	1.3%
TR	3.4%	0.2%	30.5%
NT	3.1%	2.6%	11.8%

It can be seen that running the data collection service on production servers using the *fprobe* tool required from 3.1% to 7.7% of the total CPU time. In contrast, data collection using the *custom tool* required from 0.0% to 2.6% of the CPU time. The CPU usage differs because of various external factors, such as the difference between the servers' technical potential (i.e., due to differences in the CPU and processor board architecture), intensity of traffic volume for the service, and others. Despite this, the table suggests that the custom tool (based on conntrack connections table) is a more efficient approach to collect data from the hosts.

However, data collection from network equipment is identified as preferred data collection method.

Chapter 5

Conclusions and Future work

The goal of this chapter is to summarize the results that were achieved. First, some conclusions will be drawn. Following this is discussion of the problems that were encountered. Finally, the future work section suggests what other related problems can be solved using the techniques described in this thesis.

5.1 Conclusions

Our work demonstrated that it is possible to identify service dependencies by simply observing network behaviour and processing communication patterns.

In this thesis project, we exploited the fact that Spotify's backend services could be identified in two basic ways. The main approach is using *DNS*. This is a natural approach due to fact that services use *DNS* themselves to locate other services and DNS is used to facilitate scaling and mobility. We also applied another approach - determining the service based upon the configuration management class of the host. However, this introduces some bias, because some classes do not belong to a service definition as it is defined in this thesis.

Using the collected NetFlow data we created a service topology graph, which depicts the communication relationships between different services. We compared this with a similar graph obtained by interviewing the various system owners. This helped them to notice and correct inaccuracies in the service topology graph from system owners. In addition, our topology graph has weighted edges, thus it is easy to see which edges have the most traffic volume.

We used the NetFlow infrastructure and network data collection of host technique to obtain the data for analysis. However, a lot of effort was put to investigate the more natural and efficient alternative - NetFlow data collection from network equipment. We learned the lesson that it is important consider the risk of not being able to implement a solution due to the external limits (such as organization's policies).

Besides the main goal of this thesis (comparing expected and real time service behaviour), we achieved several other results.

NetFlow infrastructure	We proposed and set up an infrastructure for collecting the NetFlow information. The suggested approach uses the SiLK suite and a deployment scenario with several collectors and a central storage and analysis machine.
Collection on hosts	We compared two methods of collecting network data on hosts. Our custom tool based on *conntrack* connection table appeared to be more efficient than using *fprobe* tool.
Analysis language	We designed and implemented a prototype of the language aimed at NetFlow data analysis. While it was a natural step to ease our own analysis, this language interpreter can be used for other important network traffic analysis tasks, such as the analysis of denial of service *(DDoS)* attacks.
Utilities for Spotify	Some tools necessary for analysis were found to be missing during the thesis project, for example an automated tool to list networks which belong to a specific Spotify *site* and a tool which could map an IP address and port number tuple to the Spotify service using DNS. These tools were implemented and can be now used by *SRE* team to ease their daily operations.

5.2 Discussions

It is clear that the definition of the service depends on the organization. Therefore, we chose to create a more flexible solution and created a separate analysis language, which allows the user to integrate other method of identifying a service. Therefore the approach could be applied in other organizations.

Due to the dynamic nature of DNS and the dynamics of the backend network infrastructure, it is practically impossible to have consistent data over a long period of time. For example, DNS entries may point to different IP addresses on consecutive days. Additional effort could be made to enable long term service related queries. These tools would have to take the information from DNS as a function of time and store this information into a database for later analysis. Alternatively, instead of triggering DNS information cache update by the analysis script, an update event may occur when a change in DNS server response to a specific query is observed by sniffing. However, a new DNS record addition and temporary removal is a more often operation than altering record to point to the different destination. The removal of DNS record does not cause problems for analysis, because services simply stop using the removed destination address. Therefore, although it is important to consider the dynamics of DNS, it is not a problem in practise.

5.3 Future Work

It would be also interesting to combine information from the configuration files with the information gathered by observing network flows. Such an approach could introduce an alternative perspective to the results.

For the purpose of analysis, we collected data from one production machine per service. Results would better reflect the actual situation if more machines would be sampled. However, it is a trade-off between accuracy and efficiency.

Section 4.2 described how a real-time service topology graph can be obtained. Provided that Spotify architects maintained a similar graph for expected dependencies, it would be possible to generate a graph showing differences between these graphs. This solution would require a strict service naming convention (i.e., rules), addition of a utility for defining such dependencies, and a consistent contribution of this information from the system owners.

It would useful to have automatic alerts, when large changes occur in communication between services located in different *sites*.

Information gathered from netflows combined with our derived language could be used for other purposes, for example examining potential DDoS attacks.

Bibliography

[1] Spotify. Background information Press Spotify. `http://www.spotify.com/about-us/press/background-info/`, 2012.

[2] P.V. Mockapetris. Domain names - implementation and specification. *Internet Request for Comments*, RFC 1035 (Standard), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966.

[3] A. Gulbrandsen, P. Vixie, and L. Esibov. A DNS RR for specifying the location of services (DNS SRV). *Internet Request for Comments*, RFC 2782 (Proposed Standard), February 2000. Updated by RFC 6335.

[4] A. Keller and G. Kar. Automated Generation of Dependency Models for Service Management. In *In Workshop of the OpenView University Association (OVUA'99)*, 1999.

[5] C. Ensel and A. Keller. Managing application service dependencies with XML and the resource description framework. In *Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on*, pages 661 –674, 2001.

[6] A. Keller and G. Kar. Determining service dependencies in distributed systems. In *Communications, 2001. ICC 2001. IEEE International Conference on*, volume 7, pages 2084 –2088 vol.7, 2001.

[7] *Web Service Dependency Discovery Tool for SOA Management Web Service Dependency Discovery Tool for SOA Management*, 2007.

[8] Sujoy Basu, Fabio Casati, and Florian Daniel. Toward Web Service Dependency Discovery for SOA Management. In *Proceedings of the 2008 IEEE International Conference on Services Computing (SCC 2008), 8-11 July 2008, Honolulu, Hawaii, USA*, pages 422–429. IEEE Computer Society, 2008.

[9] Linux Man. Auditd linux manual. `http://linux.die.net/man/7/audit.rules`, 2012.

[10] OpenSUSE documentation. The Linux auditd framework. `http://doc.opensuse.org/products/draft/SLES/SLESsecurity_sd_draft/part.audit.html`, 2012.

[11] Stackoverflow. Finding short lived TCP connections owner process. `http://serverfault.com/questions/352259/finding-short-lived-tcp-connections-owner-process`, 2012.

[12] Bousquf. Example of extracting IP addresses from auditd logs. `http://wiki.nokernel.net/linux-auditd`, 2012.

[13] Devloop. Parsing different kind of socket calls from auditd logs. `http://my.opera.com/devloop/blog/show.dml/2036593`, 2012.

[14] Iptables. Iptables tutorial. `http://www.frozentux.net/iptables-tutorial/iptables-tutorial.html#IPFILTERING`, 2012.

[15] B. Hubert. Linux Advanced Routing and Traffic Control HOWTO. `http://www.lartc.org/lartc.pdf`, 2012.

[16] IPTables tutorial. The conntrack entries. `http://www.faqs.org/docs/iptables/theconntrackentries.html`, 2012.

[17] Iptables. Iptables log options. `http://www.linuxtopia.org/Linux_Firewall_iptables/x4238.html`, 2012.

[18] B. Claise. Cisco Systems NetFlow Services Export Version 9. *Internet Request for Comments*, RFC 3954 (Informational), October 2004.

[19] Netfilter/iptables project. Libnetfilter_conntrack description. `http://netfilter.org/projects/libnetfilter_conntrack/index.html`, 2012.

[20] Dpkt. Dpkt python packet creation/parsing library. `http://code.google.com/p/dpkt/`, 2012.

[21] Cisco. NetFlow header and message format. `http://www.cisco.com/en/US/docs/net_mgmt/netflow_collection_engine/3.6/user/guide/format.html`, 2012.

[22] P. Phaal, S. Panchen, and N. McKee. InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. *Internet Request for Comments*, RFC 3176 (Informational), September 2001.

[23] Baek-Young Choi and Supratik Bhattacharyya. Observations on Cisco sampled NetFlow. *SIGMETRICS Perform. Eval. Rev.*, 33(3):18–23, December 2005.

[24] Flowd. Flowd collector website. `http://www.mindrot.org/projects/flowd`, 2012.

[25] Fprobe. Fprobe on SourceForge. `http://sourceforge.net/projects/fprobe/files/`, 2012.

[26] Nfdump. Nfdump on SourceForge. `http://nfdump.sourceforge.net`, 2012.

[27] FlowTools. Flowd collector website. `http://www.splintered.net/sw/flow-tools/`, 2012.

[28] CERT SA team. SiLK Documentation. `http://tools.netsa.cert.org/silk/index.html`, 2012.

[29] CERT SA team. SiLK Installation Guide. `http://tools.netsa.cert.org/silk/install-handbook.pdf`, 2012.

[30] Juniper Networks. Setting up J-Flow on a J-Series router. `http://kb.juniper.net/InfoCenter/index?page=content&id=KB12512&actp=RSS&smlogin=true`, 2012.

[31] Vladislav Marinov and Jürgen Schönwälder. Design of an IP Flow Record Query Language. In *Proceedings of the 2nd international conference on Autonomous Infrastructure, Management and Security: Resilient Networks and Services*, AIMS '08, pages 205–210, Berlin, Heidelberg, 2008. Springer-Verlag.

[32] Sourceforge. MatPlotLib: Python plotting documentation. `http://matplotlib.sourceforge.net`, 2012.

[33] MaxMind. GeoIP Python API. `http://www.maxmind.com/app/python`, 2012.

[34] PythonDns. DnsPython home page. `http://www.dnspython.org`, 2012.

[35] E. Koutsofios and S. North. DOT language tutorial. `http://www.phi.uu.nl/~js/graphviz/dotguide.pdf`, 2012.

[36] Graphviz. Graph visualisation software. `http://http://www.graphviz.org/`, 2012.

Appendix A

Configuring routers

This appendix describes major steps for enabling netflow data on Juniper family routers. First of all, an accepting firewall rule should be added.

```
firewall {
    family inet {
        filter sample-in {
            term default {
                then {
                    sample;
                    accept;
                }
            }
        }
    }
}
```

Later, we add a filter.

```
interfaces {
  XX {
    unit YY {
      description "Netflow export filter";
      family inet {
        filter {
          input sample-in;
        }
        address Z.Z.Z.Z/Z;
     }
   }
 }
}
```

Finally, we enable packet sampling after the record has passed the filter.

```
forwarding-options {
  sampling {
    input {
      family inet {
         rate 100;
         run-length 1;
         max-packets-per-second 1000;
      }
    }
    output {
      cflowd x.x.x.x {  // Ip address of collector
        port xxxx; // Upd port of collector
        version 5;
        no-local-dump;
        autonomous-system-type peer;
      }
    }
  }
}
```

max-packets-per-second defines limit for netflow packets (defaults to 1000), the *rate* is the denomitator of sampling rate and *run-length* sets the number of samples following the initial triggering event. In the output chain, we define the collectors' IP address and *port*, *version* indicates which netflow version we are using, and the *no-local-dump* flag indicates that we should switch off debugging.

Appendix B

Measurements

This appendix gives full details of measurements of two data collection methods.

Table B.1: *Measurement of impact on server performance caused by data collection*

	fprobe			custom tool		
Service	tool CPU	total CPU	fraction	tool CPU	total CPU	fraction
A	11.0%	294.0%	3.7%	2.3%	284.0%	0.8%
B	8.5%	180.0%	4.7%	5.0%	176.7%	0.0%
C	3.4%	75.7%	4.5%	1.8%	74.4%	2.4%
D	18.7%	327.0%	5.7%	3.2%	330.0%	1.0%
E	14.2%	185.5%	7.7%	0.2%	175.8%	0.1%
F	1.1%	26.7%	4.1%	0.1%	26.0%	0.4%
G	8.1%	238.4%	3.4%	0.4%	228.2%	0.2%
H	2.1%	67.1%	3.1%	0.4%	65.4%	0.6%

Appendix C

Language constructs

The following table gives a summary of available language constructs.

Statement	**Description**	**Example usage**
import	import custom Python module. The interpreter looks for file with ’py’ extension in current directory.	import ’geo’;
filter	filters nodes that match certaing criteria from the context.	filter geo(’LT’) or geo(’LV’);
group	groups each node in the context using the specified grouper.	group servicegrouper(’dip’) as g1;
gfilter	filters groups themselves.	gfilter bps() >mb(5);
emit	emits values calculated for the current context.	emit (sum(“bytes”));
option	set one of the optional parameters for plot: *title* (string), *xlabel* (string), *ylabel* (string), *obfuscate* (int), *angle* (int)	option title ’the chart’;

Appendix D

Language built-in functions

The following table documents all currently available functions in our NetFlow data analysis framework.

Function	Description	Example usage	Module
count()	get number of nodes (connection) in context	gfilter count() >100;	standard
sum(attr)	converts attribute *attr* to int for each node in context and get sum	gfilter sum('bytes') <10;	standard
packets()	shorthand for call *sum('packets')*	gfilter packets() <100;	standard
bytes()	shorthand for call *sum('bytes')*	gfilter bytes() >0;	standard
bps()	get average number bytes per second for context, i.e. shorthand for *sum('bytes') / seconds*	gfilter bps() >10;	standard
kb(c)	get bytes from kB, an equivalent of *c * 1024*	gfilter bytes() >kb(10);	standard
mb(c)	gets bytes from mB, an equivalent of *c * 1024 * 1024*	gfilter bytes() >mb(1);	standard
diff(attr)	gets number of different values of given attribute among the context nodes	emit (diff('sip'));	standard
log(c)	gets logarithm (base e) on argument *c*. It is useful when plotting logarithmic plots.	emit (log(count()));	standard
attribute(attr)	groups flows by node attribute *attr*;	group attribute('sip');	standard

time(bin, start, ..)	groups flows into timebins, where *bin* is one time bin size (default five minutes), *start* and *end* (default is currentime) are timestamps for interval in interest, *duration* (default one hour) is interval length, and *bins* is number of bins (default is 20). Enough parameters should passed to command in order to correctly identify interval of interest.	group time(bins=100)	standard
geo(by, mode)	groups nodes by geographical location. Parameter *by* defines the attribute that will be used for grouping: 'sip' (default) or 'dip'. Parameter *mode* defines what grouping mode should be used: either fetching full country name (value 'name', default) or two letter code ('code').	group geo('dip');	geo
fscc()	filters nodes by the source country code	filter fscc('LT');	geo
fdcc()	filters nodes by the destination country code	filter fdcc('LT');	geo
classgrouper(attr)	groups nodes by configuration management class name. Parameter *attr* is attribute to use: 'dip' (default) or 'sip'	group classgrouper('sip');	services
servicegrouper(attr)	groups nodes based on DNS information. Parameter *attr* is attribute to use: 'dip' (default) or 'sip'	group servicegrouper('dip') as g1;	services

TRITA-ICT-EX-2012:63