Flow-Based Dissection of Network Services

Dominik Schatzmann
ETH Zurich
schatzmann@tik.ee.ethz.ch

Wolfgang Mühlbauer
ETH Zurich
muehlbauer@tik.ee.ethz.ch

Bernhard Tellenbach
ETH Zurich
tellenbach@tik.ee.ethz.ch

Simon Leinen
Switch
simon.leinen@switch.ch

Kavé Salamatian
Université de Savoie
kave.salamatian@univ-savoie.fr

ABSTRACT

The unprecedented success story of the Internet is largely due to rich and constantly emerging applications such as online social networks, video streaming, etc. To characterize the Internet and its usage, high-level metrics such as traffic volume or topology-related measures have been widely used in the past. However, researchers and network professionals still lack concepts to capture Internet services.

Our approach for monitoring applications and services is to be agnostic. Starting at the granularity of flow-level information, we propose a system that can efficiently detect communication end points that offer service to multiple clients. In this technical report, we dive into some details with respect to the data structures and data processing techniques that are needed to achieve our goals.

1. INTRODUCTION

Most likely, there is no system with a similar degree of diversity in its usages as the Internet. The history of the Internet during the past 40 years is characterized by the emergence of new and fast growing applications and services. Online social networks [10], video streaming [30], AJAX-based text processing, spreadsheet, or webmail applications [29], are just some examples.

This rapidly changing environment makes it very challenging for network professionals (operators, engineers, application designers) to anticipate how the network and its resources will be used. Continuous observations and measurements are therefore required to monitor the network. The research community is still facing the challenge to relate observable, high-level metrics such as traffic volume [5, 8, 12], or topology-related measures (e.g., [17, 23, 24, 28]), with higher-level concepts like applications and services.

Nevertheless, where observable metrics like packet rates, etc., can be defined through objective means, the concepts of application or services are more difficult to capture, and are unfortunately ill-defined. To illustrate this difficulty, one can consider the case of an HTTP flow transferring a streaming video from a video-conference. Should this flow be categorized as a HTTP flow, or as video streaming, or as a video conference one? Similar issues exist for service definition. Being aware of these issues we adopt in this paper a different approach.

In the spirit of Estan et al. [9], we start from the granularity of flow-level information as observed at the border of a large ISP with more than 2 million internal hosts. Our approach for monitoring applications and services is to make minimal assumptions, and let the data speak by itself. For this reason and to avoid using the overloaded terms "application" and "service", we define the notion of a *server socket*: a server socket is identified by a tuple of IP address, port address, and protocol number, and acts as a concentrator, i.e., it communicates with multiple sockets. Alike, we refer to a host on which resides at least one server socket as a *server host*.

Key to our detection of server sockets is a greedy approach. We first identify server sockets that communicate with a high number of communication end points, and then turn to the remaining sockets. However, we do not restrict ourselves to the top server sockets only. Importantly, our system also identifies server sockets that run on high ports and that are potentially not involved in a lot of traffic, hence frequently being overlooked. Finally, our system remembers already detected server sockets to cope with the high data rates of $14-40K$ NetFlow records per second. Our contributions are threefold.

Previous measurement studies [6,12] have mainly reported about network applications or services in the form of aggregate data, e.g., "what amount of traffic is P2P?". Contrary to this, our objective is to identify and analyze the communication end points that offer service to a number of clients, and not to study exchanged traffic per se. In particular, we can detect services that are frequently overlooked (low-traffic applications on high ports), but, that as a whole turn out to be more relevant than previously thought. In total, our approach provides a detailed view about server sockets, and about their locations in terms of hosts, subnets, Autonomous Systems (ASes), or even countries. This can help network operators to anticipate how the network and its resources will be used [1, 2, 19].

To sum up, we have implemented a system that can process large-scale, flow-level data sets for the purpose of detecting server sockets. Our agnostic approach takes into account both TCP and UDP services, does not require TCP/SYN flags as used by Bartlett et al. [4], copes with noise from scanning [3], and with low resolution of NetFlow timestamps. Our proposed techniques allow to process our 5-day trace from 2010 within less than 4 days, and our 5-day trace from 2003 within one day, suggesting that processing in real time is feasible.

The rest of this paper is structured as follows: Section 2 describes our data sets and provides a general overview of our approach. Section 3 explains the individual steps of our data processing. Finally, we review related work in Section 4 and conclude in Section 5.

TIK Report, ETH Zurich, 2011.

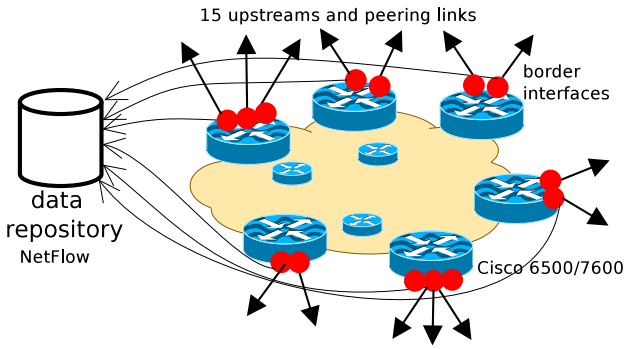

Figure 1: Data collection.

2. GENERAL APPROACH

Our goal is to capture network applications and services, and to study long-term trends in their usages. In Section 2.1 we explain the used data set and justify why our data is perfectly suited for our needs. Section 2.2 gives a high-level overview of our approach, and summarizes its key ideas.

2.1 Data Sets

We rely on data collected at SWITCH [25], an ISP that mainly connects research labs, universities, and government institutions to the Internet. Importantly, we point out that SWITCH is not just a small network. According to student registration and employee statistics, the estimate number of actual users of our network amounts to approximately 250,000 in 2010, and 200,000 in 2003. The IP address space that is announced via BGP to the Internet currently covers approximately 2.4 million IP addresses. Since this number has remained relatively stable over the years, we conclude that the network itself has not undergone any significant changes in its user base. Hence, it is ideally suited for studying network services and general changes in their usage.

Traffic information has been collected in the form of flow-level data since 2003 until today. We record *all* flows that cross any external interface of any border router, thus obtaining all traffic that the studied network exchanges with the Internet. To this end, we capture unsampled NetFlow data at all six border routers with a total of 15 upstream and peering links in 2010. To cope with high traffic rates, we have to rely on hardware-based flow collection [15], using NetFlow in its version 9. Due to limitations of the used hardware, information about TCP flags ("cumulative OR of TCP flags") is not available. Finally, we store the obtained flow records at a central data repository. Figure 2.1 summarizes our data collection.

For our studies we extract from our data archives one 5-day trace (Monday, 00:00 UTC until Friday, 24:00 UTC) from 2010. We choose the first full working week of November, which is in the middle of the fall term. The large-scale nature of our measurements is underlined by high peak rates of more than 80,000 flows per second, 3 million packets per second, and more than 20 Gbit/s of traffic. In addition, the traffic characteristics show as expected strong daily patterns [26].

2.2 Methodology

The unprecedented success story of the Internet is mainly due to the high variety of offered network applications or services, ranging from classical web, mail, or FTP servers to P2P nodes, online social networks, video streaming etc. Unfortunately, both the term "application" and "service" is overloaded, and not concisely defined. Therefore, we define the following two abstractions:

server socket: We rely on sockets as the fundamental abstraction for describing network services or applications. A socket is uniquely identified by IP address, protocol number (e.g., 6 for TCP, 17 for UDP), and TCP/UDP port number. A server socket is a socket that according to our flow-level data acts as a concentrator, i.e., it communicates with multiple sockets.

server host: A host on which resides at least one server socket is referred to as a server host.

By using these definitions to describe network applications and services, we make minimal assumptions about the specific type of a service or application (e.g., is it a web server or is it Skype super node?). Yet, we ensure that we capture those "services" we are interested in, namely communication end points that offer service to *multiple* other communication end points (sockets). Such an approach can detect server sockets for classical services (web, e-mail, FTP, SSH, etc.), but also for P2P-like communication (P2P super nodes, Skype, etc.). Only a very limited class of network applications shows a completely symmetric communication pattern (e.g., a BGP session, direct communication between two P2P nodes that are not supernodes, NNTP inter-server feeds). Importantly, our definitions do not exclude services that are frequently overlooked such as services running on high ports, or services that attract or induce only low traffic volumes.

More details on our heuristics are part of Section 3. Here, we already point out that there exist many other challenges related to the detection of server sockets. The sheer size of our data requires careful data processing to eliminate noise from scanning, to cope with limited information, e.g. in the absence of TCP SYN flags [15] or imprecise timing information, or simply to correctly merge unidirectional to bidirectional flows. To this end, we believe that our work can provide useful hints for any type of research based on flow-level data. In contrast to most work from the past, our detection of server sockets is completely *passive*. It works even for large networks consisting of 2.4 million IPs if data structures and processing are efficiently implemented.

Overall, our approach to extract server sockets is guided by three key ideas:

Server sockets are concentrators: As already mentioned above, server sockets offer "service" to multiple sockets. Analyzing collected flow information for a certain time period, we expect to observe that multiple distinct sockets talk with a socket that represents a "service". In Section 3, we will leverage the number of sockets that communicate with a given socket as indicator for the existence of a server socket. As we will show, such an approach generally allows to unambiguously discriminate a "client" and a "server" side.

Greedy approach: Essentially, the input to our data processing is a tremendously large number of flow records. We build up data structures that allow to first extract those server sockets with the highest number of associated sockets. After its detection, a server socket is removed from our data structures. Internally, we then update all counters that track the number of associated sockets for each given socket. Then, we continue our search for server sockets. Such a greedy approach detects first the important server sockets, but continues to search for server sockets that could be regarded as less relevant because only a few sockets connect with them. The general details of our detection algorithms are part of Section 3.

Recalling detected server sockets: Apparently, many network applications such as a web server are stable and are continuously being accessed by some Internet users. We leverage this to scale our detection heuristics. After a server socket has been extracted, we store its IP address, port number, protocol information in a separate data structure (registry). Hence, during later time intervals, we do not necessarily have to reclassify these server sockets, but can focus on newly appearing network applications. We believe that such a "stateful" approach can be beneficial for a wide range of applications in the field of network measurements.

The combination of these ideas enables us to perform a large-scale study of network applications and their usage across a time period of 8 years. In addition, to the data collection efforts and data processing techniques, one major contribution of our work lies in the detection of *high port* network applications. In particular, P2P server nodes such as a BitTorrent or Skype super nodes do use such port ranges (above 1024).

Finally, we point out that our approach allows to detect network applications both inside and outside the studied network. Network operators can implement the ideas of this paper to obtain a application-level map of their network. For example, this is useful to anticipate how the network and its resources will be used, and to tailor the peering and network infrastructure to the needs of the network users.

3. IMPLEMENTATION

The implementation of our data processing to detect server sockets includes six major components, see Figure 2. The *data parser* obtains unidirectional flow data from the central data repository described in Section 2.1. It decompresses bzip2 files and parses NetFlow version 9. Afterwards, a *preprocessing* step removes some flows from the data stream that are not needed for the detection of server sockets. For a limited time, we keep the remaining flows in the *connection cache*. If we find for a unidirectional flow a corresponding reverse flow, we merge both into a connection and store them together in the connection cache. Any entry in this cache is uniquely identified by the IP address, protocol number, and port number of both the internal and the external host.

Up to this point, data processing is stream-based, i.e., information about incoming flows is continuously inserted into the connection cache. However, further processing is clock-based, since the detection of server sockets requires to study a large set of flows across many connections within a certain time interval, see Section 2. Whenever a certain time interval t has elapsed, we extract all connections from the connection cache that have been observed during the elapsed time interval t. Then, we perform an additional step of *filtering* on these flows, mitigating the impact of scanning activities and eliminating general noise.

Finally, we identify from the remaining set of connections the server sockets, i.e., the side of a connection that actually offers the "service", see Section 2.2. To this end, our data processing first does a lookup in the *registry*. This component recalls already identified server sockets from previous time intervals, which are identified by the three-tuple of IP address, port, and protocol number. For connections after the filtering step we distinguish between two cases: if there is already an entry for one side of the connection in the registry, the server socket is already known and we are done. Otherwise, we forward the connection to the *socket detection* engine. There, we adopt a greedy heuristic as outlined in Section 2.2 to identify server sockets. After the detection of new server sockets, we add them to the registry.

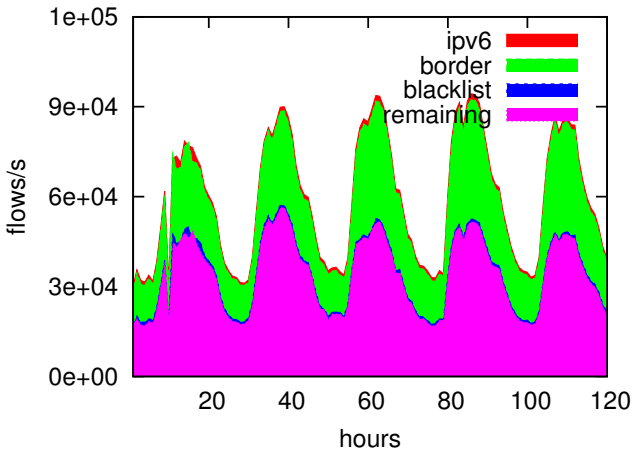

Figure 3: Data preprocessing

In the following, we will provide more details about the implementation of the individual components. Section 3.1 summarizes our data preprocessing, Section 3.2 explains our data structure for the connection cache, Section 3.3 discusses how to mitigate the impact of noise and scanning, and Section 3.4 introduces our heuristics to detect new server sockets. Finally, Section 3.5 discusses the performance of our system.

3.1 Preprocessing

Altogether, we perform three filtering steps: first, we currently remove IPv6 flows . Even today, its traffic volume is still far below 10% (June 2011). Yet, we point out that our techniques can be directly applied to IPv6 data without any major modifications. Second, we only keep those flows that either originate or terminate inside our network. In particular, traversing flows or network-internal flows are removed. By doing so, we ensure that if traffic is bidirectional between a pair of hosts, we always must observe both directions. Finally, we filter out a flow if its source or destination is on our manually generated prefix blacklist. For example, we ignore PlanetLab hosts and eliminate bogon IP addresses based on the bogon prefix list that we obtained for the year 2010 [16].

The plot of Figure 3 shows for our 2010 trace the flow rates averaged over 15 minutes for IPv6 (*IPv6*), internal and traversing flows (*border*) and flows from or to hosts found on our *blacklist*. While the *IPv6* and *blacklist* filters only remove a limited amount of data, the *border* filter eliminates generally more than 40% of all flows. In total, there remain around 50% of all flows after this step.

3.2 Connection cache

After the preprocessing step, the unidirectional flows arrive at the connection cache. One central task of this component is to merge unidirectional into bidirectional flows if the source identifiers (IP address, port number) of a flow match the destination identifiers of another flow, and vice versa. After this step, we generally talk about *connections* irrespective of whether we have found flows for both directions, or only for one direction. We maintain a hash-like data structure that, for observed connections identified by IP addresses, protocol number, and application ports, stores and updates information that is relevant for further analysis. This includes packet counts, byte counts, and the time when the connection was active (start and end time).

Data processing up to the connection cache is stream-based. Incoming flow information is continuously parsed, filtered, and fi-

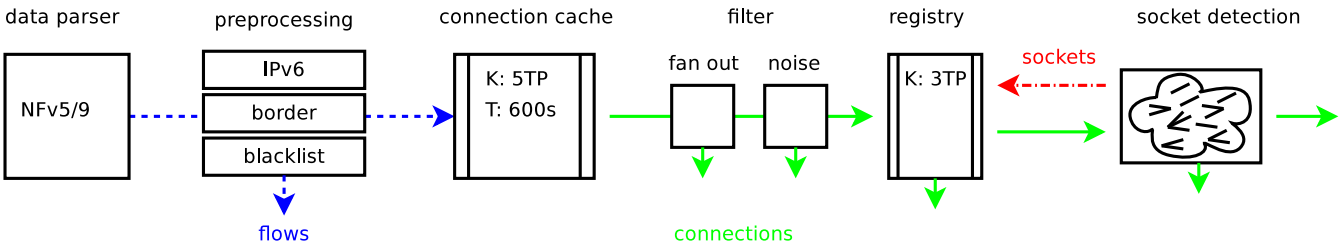

Figure 2: Implementation overview.

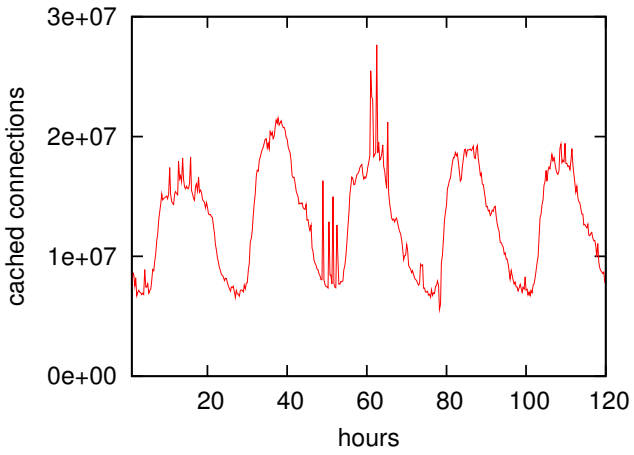

Figure 4: Size of the connection cache.

nally inserted into the connection cache. Subsequent data processing is clock-based, because we need to consider a larger set of connections across a certain time interval t to detect server sockets, see Section 3.4. Therefore, we partition the timeline into intervals of t minutes, and proceed with our data processing whenever such a time interval has elapsed. For t we choose 15 minutes, which makes it likely to observe at least some flows for any active server socket. Every 15 minutes we extract from the cache connections that have been active during the preceding 15 minutes time interval, delete them from the cache, and pass them on to the filtering unit described in Section 3.3.

Efficient memory handling ensures that data does not have to be copied when "forwarding" connections from the connection cache to the subsequent units for noise elimination or server socket detection.

Figure 4 shows for our 2010 trace the number of connections that are stored in the connection cache. We find that the cache generally contains between 7 and 28 million connections. As expected the curve follows daily patterns [26], but also shows spikes. We speculate that such rapid increases in the number of stored connections are due to scanning activities.

3.3 Noise Elimination

Whenever a time interval of 15 minutes has expired, flows that have been active during the preceding interval are removed from the cache and undergo an additional step of filtering, see Figure 2. Our goal is to eliminate noise that could impede the detection of server sockets.

First, we try to cull potential scanning activities. To this end, we borrow the key idea from Allman et al. [3]. Scanning attempts generally do not result in established connections, and are not limited to individual hosts or ports. The pseudo code listing of Algorithm 1 summarizes our approach. We examine the fanout of hosts, i.e., the number of 2-tuples (IP address, port number) that a host tries to access. More precisely, we attribute two counters to each host: fan_{good} reflects the number of bidirectional TCP (UDP) connections with more than 2 (1) packet(s) in both directions in which this host participates. Similarly, fan_{bad} represents the number of unidirectional flows where this host is listed as source. Following the result of Allman et al. [3], we finally classify a host as a scanner if it has fan_{bad} of at least 4 and if fan_{bad} is at least twice fan_{good}.

Algorithm 1 Scanning detection

```
/* fan_good and fan_bad count fanout;
counters initialized to 0 **/

for all connections |c| of connection cache do
  if (bidirect. TCP and > 2 in/out pckts) then
    fan_good[c.ipaddr_in]++
    fan_good[c.ipaddr_out]++
  else if (bidirect. UDP) then
    fan_good[c.ipaddr_in]++
    fan_good[c.ipaddr_out]++
  else if > 0 out packets and 0 in packets then
    fan_bad[c.ipaddr_in]++
  else if > 0 in packets and 0 out packets then
    fan_bad[c.ipaddr_out]++
  end if
end for
for all |ip| IP addresses in the connection cache do
  if (fan_bad[ip] > 4 and fan_bad[ip] > 2 * fan_good[ip]) then
    ⇒ SCANNING
  end if
end for
```

Connections that are associated with a detected scanner are removed from further data processing. Figure 5 shows that adopting this strategy reduces the incoming connections per second by 22% on average. More importantly, we find that many of the spikes disappear, indicating the efficiency of our scan detection heuristic.

We point out that the discussed heuristics do not eliminate connections with low traffic volume such as pure TCP 3-way handshakes. However, we are mainly interested in connections where a minimum amount of user data is exchanged. Therefore, a second filtering unit eliminates all (i) unidirectional TCP connections, (ii) TCP connections with less than 4 packets, and (iii) UDP connections with less than one packet in both directions. As shown in Figure 5 ("noise"), this decreases the connection rate per second by 1,300 on average.

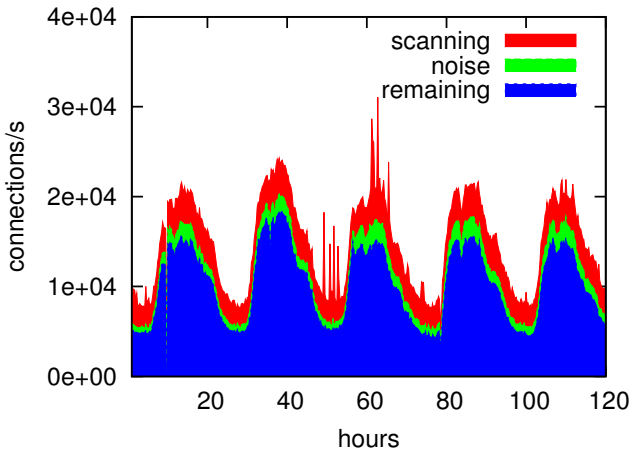

Figure 5: Filtering.

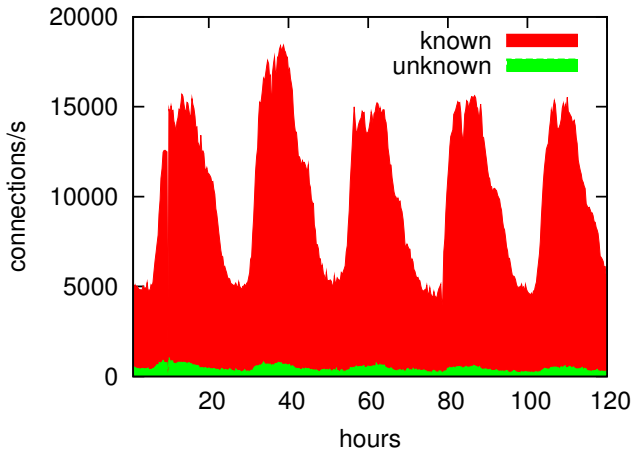

Figure 6: Registry – known vs. unknown server sockets.

3.4 Detection of Server Sockets

Now, we discuss our heuristics to detect server sockets from *all* connections that remain after the filtering step. For this purpose, we can neither rely on timing information nor on TCP flags to determine the originator of a connection. However, our solution is based on flow-level data, allowing for large-scale characterization of network applications. Importantly, we do not neglect server sockets that use UDP as transport protocol, or that run on high ports.

As shown in Figure 2 and discussed in Section 2, we implement a *registry* component that remembers already detected server sockets and passes only those connections to the detection engine that cannot been associated with any already known server socket. After a certain warm-up phase, the registry reduces the number of connections per second that need to be forwarded to the server detection unit by 95%. Evidently, keeping and recalling already detected server sockets, significantly decreases the computational load for server socket detection.

The remainder of this section will introduce the heuristics that we use to detect new server sockets given a set of still unassigned connections.

Our approach is guided by two key ideas. First, we regard 3-tuples (IP address, port number, protocol number) as hot candidates for being a server socket if according to our connection data these sockets communicate with a large number of other endpoints (again, identified by IP address, port, protocol number). Second, we adopt a greedy approach, see Section 2 that first selects the 3-tuples as server sockets with the highest number of associated "client" 3-tuples. We iteratively remove first the "important" server sockets, delete all their associated connections, and then continue with the server socket detection based on the remaining set of connections.

The pseudo code listing of Algorithm 2 illustrates in a simplified way how our approach works. Initially, we compute two separate lists, one for all 3-tuples that are inside our studied network, and the other for all 3-tuples that are outside our network. Moreover, we sort both lists by the number of "client 3-tuples" with which a 3-tuple communicates (degree $deg(x)$). Within the main lookup, the first while loop starts extracting all 3-tuples from inside our network until the maximum number of "client 3-tuples" falls below the maximum number of "client 3-tuples" for the external candidates. Accordingly, the second inner loop extracts server sockets that are outside of our network. Every time after choosing a server socket, we remove all connections associated with the detected server socket, and recompute only the list SS_{in} or SS_{out}, respectively. Overall, the extraction process continues until all remaining internal and external candidates have a degree of less or equal than 2.

Algorithm 2 Detection of server sockets.

```
compute list SS_in {int. sockets sorted by # ext. clients}
compute list SS_out {ext. sockets sorted by # int. clients}

/* deg(x) returns the number of clients for a socket **/

while deg(SS_out[0])> 2 or deg(SS_in[0])> 2 do
  while (deg(SS_in[0]) > deg(SS_out[0])) do
    ss = SS_in[0] {classify ss as internal server socket}
    remove ss from SS_in
    update deg() for all entries of SS_in
  end while
  while (degree(SS_out[0]) >= degree(SS_in[0])) do
    ss = SS_out[0] {classify ss as external server socket}
    remove ss from SS_out
    update deg() for all entries of SS_out
  end while
end while
```

3.5 System Performance

Finally, we discuss the performance of our server detection techniques. Figure 6 reveals that 96% of all connections that enter the server detection component per second can be assigned to a server socket. Only 4% need to be classified by the server detection unit. By design, see Algorithm 2, the remaining connections are associated with 3-tuples that talk with less than three other "client" 3-tuples. In addition to the expected daily patterns in the connection rates, we see that the number of unclassified connections steadily decreases during the 5 days of our trace, meaning that more and more connections can already be assigned to a server socket by the registry.

In contrast to connection rates, Figure 7 displays the number of newly detected sockets per 15-minute time intervals across the runtime of our 5-day trace from 2010. In general, most new server sockets (peaks of 5,000 to more than 16,000) are located outside our studied network and use TCP, followed by external UDP sockets, internal UDP sockets, and internal TCP sockets. Generally, we

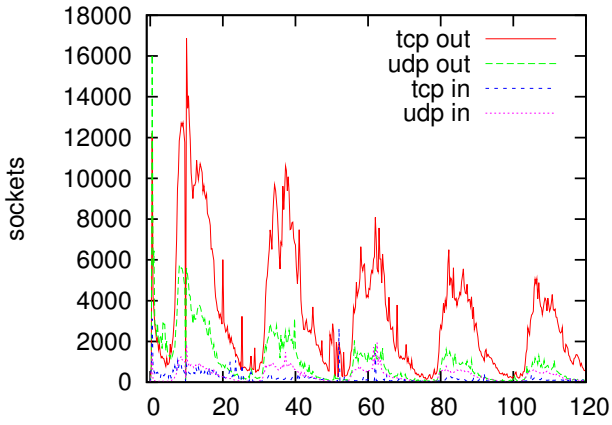

Figure 7: New server sockets by classes.

find a higher number of new sockets in the beginning of our trace, since only few server sockets will be known to the registry in the warm-up phase.

So far, this section has illustrated the challenges of processing large-scale traffic data with peak rates of more than 20 Gbit/s. Therefore, we had to carefully design our data structures and implementation. Overall, we have implemented more than 12,000 lines of code. While all performance-critical parts of our code are in C++ , we sometimes also rely on Ruby to instrument our measurements and to profile our system. On a 2.2 GHz AMD Opteron 275 with 16GB memory[1], we can process our 5-day trace from 2010 within 91 hours. Older traces are considerably faster, with the 2003 trace finishing with 20 hours. The memory required by our approach is dominated by the number of connections in the connection cache (see Figure 4) and the data stored per connection. Typically, we need around 128 bytes per connection and 9 GB of memory in total.

These numbers clearly suggest that our approach can come close to real-time processing. This holds in particular, if we do not have to use shared hardware resources as for this paper. Therefore, we can picture exciting potentials of our measurement approach and the proposed data processing techniques if they are permanently applied to a real network. For example, they can provide network operators with an overview about network applications, their usages, and their location.

4. RELATED WORK

Our paper touches upon two important aspects of networking research and network operation: (i) discovery of network services, (ii) monitoring and troubleshooting of networks and their usage, (iii) characterization of Internet traffic and applications.

Several proposals have been made to detect network services, many of them relying on active probing, e.g., [4, 13] instead of passively collected flow-level data as in our paper. Another line of work has tried to compare the powerfulness of passive vs. active service discovery [4, 27]. Although detection of well-known services can be "straightforward" as stated by Bartlett et al. [4], our paper clearly reveals the challenges (e.g., missing TCP flags and imprecise timer information) for a more large-scale data set, and, importantly, studies high-port applications. In contrast to a packet-level solution [27], collecting flow-level data can scale well for large ISPs with more than 2 million internal hosts. However, due to the limited nature of flow-level information, it becomes more challenging to process the data and to draw conclusions [20, 22]. Possibly closest to this paper is the work by Karagiannis et al. [11] who classify traffic flows according to the applications that generate them. Here, we borrow the idea of detecting the sources of traffic. Unfortunately, Karagiannis et al. only report results for a user population of 20,000 and completely ignore port numbers, although we have shown that these are beneficial to better understand application characteristics.

We believe that the results of this paper can pave the way for more sophisticated network monitoring and troubleshooting approaches. The tool suite Flowscan [18] requires the collection of flow-level data, entails a high performance database for storing flow data [21], and can be combined with powerful visualization tools, e.g., [7]. Yet, automatic detection of (new) services is not possible per se. Commercial solutions such as Nagios [14] can be used for monitoring, alerting, reporting, maintenance, and planning tasks and are strongly based on SNMP or periodic probing. They also do not allow to automatically detect new network applications.

5. CONCLUSION

The main objective of this work is to outline novel ways to capture network applications and services. Starting at the granularity of flow-level information, we have proposed a system that can efficiently detect communication end points that offer service to multiple clients. In this technical report, we dived into some details with respect to the data structures and data processing techniques that are needed to achieve our goals.

Overall, we believe that there is a lot of potential for future work. In particular, we plan to actually deploy our system at the studied network. This will be beneficial for the network operators to understand and anticipate how the network and its resources will be used. To this end, we intend to refine and integrate our implementation into existing tracing tools (e.g., nfdump), to provide summary reports that allow network operators to quickly get an overview of running services and applications, and to release our code to the public.

6. REFERENCES

[1] *Green Networking '10: Proceedings of the first ACM SIGCOMM workshop on Green networking* (New York, NY, USA, 2010), ACM. 533109.

[2] AGGARWAL, V., FELDMANN, A., AND SCHEIDELER, C. Can ISPs and P2P systems co-operate for improved performance? *ACM CCR 37*, 3 (2007), 29–40.

[3] ALLMAN, M., PAXSON, V., AND TERRELL, J. A Brief History of Scanning. In *Proc. ACM IMC* (2007).

[4] BARTLETT, G., HEIDEMANN, J., AND PAPADOPOULOS, C. Understanding Passive and Active Service Discovery. In *Proc. ACM IMC* (2007).

[5] BHARTI, V., KANKAR, P., SETIA, L., GÜRSUN, G., LAKHINA, A., AND CROVELLA, M. Inferring invisible traffic. In *Proc. ACM CoNEXT* (2010).

[6] BORGNAT, P., DEWAELE, G., FUKUDA, K., ABRY, P., AND CHO, K. Seven Years and One Day: Sketching the Evolution of Internet Traffic. In *Proc. IEEE INFOCOM* (2009).

[7] Cacti – the complete rrdtool-based graphing solution. `http://www.cacti.net/`.

[8] CHANG, H., JAMIN, S., MAO, M., AND WILLINGER, W. An Empirical Approach to Modeling Inter-AS Traffic Matrices. In *Proc. ACM IMC* (2005).

[1]Machine was shared with user users.

[9] ESTAN, C., SAVAGE, S., AND VARGHESE, G. Automatically Inferring Patterns of Resource Consumption in Network Traffic. In *Proc. ACM SIGCOMM* (2003).

[10] Facebook. http://www.facebook.com/.

[11] KARAGIANNIS, T., PAPAGIANNAKI, K., AND FALOUTSOS, M. BLINC: Multilevel Traffic Classification in the Dark. In *Proc. ACM SIGCOMM* (2005).

[12] LABOVITZ, C., IEKEL-JOHNSON, S., MCPHERSON, D., OBERHEIDE, J., AND JAHANIAN, F. Internet Inter-Domain Traffic. In *Proc. ACM SIGCOMM* (2010).

[13] LEONARD, D., AND LOGUINOV, D. Demystifying Service Discovery: Implementing and Internet-Wide Scanner. In *Proc. ACM IMC* (2010).

[14] Nagios. http://www.nagios.org/.

[15] Introduction to Cisco IOS NetFlow - A Technical Overview. http://www.cisco.com/en/US/prod/collateral/iosswrel/ps6537/ps6555/ps6601/prod_white_paper0900aecd80406232.html/.

[16] Team Cymru Community Services. http://www.team-cymru.org/Services/Bogons/.

[17] OLIVEIRA, R., PEI, D., WILLINGER, W., ZHANG, B., AND ZHANG, L. In Search of the Elusive Ground Truth: the Internet's AS-Level Connectivity Structure. In *Proc. ACM SIGMETRICS* (2008).

[18] PLONKA, D. FlowScan: A Network Traffic Flow Reporting and Visualization Tool. In *Proceedings of the 14th USENIX conference on System administration* (2000).

[19] POESE, I., FRANK, B., AGER, B., SMARAGDAKIS, G., AND FELDMANN, A. A Brief History of Scanning. In *Proc. ACM IMC* (2007).

[20] QUAN, L., AND HEIDEMANM, J. On the Characteristics and Reasons of Long-lived Internet Flows. In *Proc. ACM IMC* (2010).

[21] RRDtool – logging & graphing. http://oss.oetiker.ch/rrdtool/.

[22] SCHATZMANN, D., MÜHLBAUER, W., SPYROPOULOS, T., AND DIMITROPOULOS, X. Digging into HTTPS: Flow-Based Classification of Webmail Traffic . In *Proc. ACM IMC* (Melbourne, Australia, 2010).

[23] SHERWOOD, R., BENDER, A., AND SPRING, N. DisCarte: A Disjunctive Internet Cartographer. In *Proc. ACM SIGCOMM* (2008).

[24] SPRING, N., MAHAJAN, R., AND WETHERALL, D. Measuring ISP Topologies with Rocketfuel. In *Proc. ACM SIGCOMM* (2002).

[25] The Swiss Education and Research Network (SWITCH). http://www.switch.ch.

[26] THOMPSON, K., MILLER, G., AND WILDER, R. Wide-Area Internet Traffic Patterns and Characteristics. *IEEE Network 11* (1997), 10–23.

[27] WEBSTER, S., LIPPMANN, R., AND ZISSMAN, M. Experience Using Active and Passive Mapping for Network Situational Awareness. In *Proceedings of the Fifth IEEE International Symposium on Network Computing and Applications* (2006).

[28] YOSHIDA, K., KIKUCHI, Y., YAMAMOTO, M., FUJII, Y., NAGAMI, K., NAKAGAWA, I., AND ESAKI, H. Inferring POP-Level ISP Topology through End-to-End Delay Measurement. In *Proc. ACM PAM* (2009).

[29] Google. http://www.google.com/.

[30] YouTube. http://www.youtube.com/.