Technische Universität München
Department of Informatics

Master's Thesis in Informatics

A Scientific Workbench for Empirical Network Research

Jan Seeger

Technische Universität München

Department of Informatics

Master's Thesis in Informatics

A Scientific Workbench for Empirical Network Research

Eine wissenschaftliche Workbench für empirische Netzwerkforschung

Author	Jan Seeger
Supervisor	Prof. Dr.-Ing. Georg Carle
Advisor	Dipl.-Inf. Johann Schlamp, Dr. Ralph Holz
Date	July 31, 2015

Informatik VIII
Chair for Network Architectures and Services

I confirm that this thesis is my own work and I have documented all sources and material used.

Garching b. München, July 31, 2015

Signature

Abstract

This work describes the implementation of a network data analysis system for analysing existing network research data. This project's aim is the implementation of a system that allows analysing existing research data regardless of storage system or format.

For this, we implement a system storing data in its original format, and allowing targetted access to that data via a centralized database and custom-made access components. Additionally, the system supports internal custom preprocessing of data, the automation of repeated experiments, a web interface for ad-hoc analysis of data as well as a python module for programmatic analysis of data.

We then evaluate the implemented system's performance and execute a simple analysis, to provide an example for usage of the system.

Zusammenfassung

Diese Arbeit dient der Implementierung eines integrierten Analysesystems für vorhandene Netzwerkdaten. Ziel ist es, ein System zu implementieren, welche die Verwertung von Netzwerkdaten unabhängig von Speicherung und Format erlaubt.

Hierzu wird ein dezentrales System implementiert, welches Daten in ihrem originalen Speichersystem aufbewahrt, und mithilfe eines zentralen Index und speziellen Zugriffskomponenten einen gezielten Zugriff auf diese Daten erlaubt. Ausserdem unterstützt das System die interne Vorverarbeitung von Daten, die Automatisierung von wiederholten Experimenten, ein Web-Interface für Ad-Hoc-Analyse von Daten sowie ein Python-Modul zur programmgesteuerten Analyse der Daten.

Das implementierte System wird dann auf seine Performance untersucht, und eine einfache Analyse durchgeführt, um ein Beispiel für die Verwendung des Systems zu zeigen.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today's internet research results in huge amounts of data being kept in research institutes. Additionally, more and more data is constantly being generated. However, available analysis technology is not keeping pace with this rapid growth of measurement data. The commodification of such measurement data (driven by faster networks and better scanning technology) makes it more important than ever to research avenues for combined and centralized data analysis.

To illustrate the point more thoroughly, we will follow the course of a single research experiment and the data it generates. An experiment is executed, generating a homogeneous mass of data. During the course of research, this data is analyzed with custom tools and methodologies. Once the experiment is completed, the data is often kept readily available, in order to facilitate reproduction of experiment results, or with the goal of eventual reuse. However, while the stored data may be readily available to the original researcher, access by other scientists is often considerably more difficult, as the custom programs, analysis methods and scripts used for the original experiment are not available publicly. Examples for this are database schematics whose semantics are not clear, binary files used for storage whose structure is not known, and missing custom scripts for conditioning the experiment data.

In general, data from such networking experiments can rarely be reused for additional research beyond that of the original experimenter. Additionally, it is difficult for researchers to gain access to the data, thus making research results hard to reproduce.

We argue that facilitating easier access to research datasets would allow additional research to take place on existing data. Existing data can also be used to extend the results of existing research, and allow reproduction of existing research.

To allow this, this work will design, implement and evaluate a system termed "Empirator". The Empirator system aims to facilitate research on combined networking experiment data from different sources, while supporting a wide variety of data, and allowing easy retrieval, selection and experimentation on that data.

However, implementing such a system is challenging. Firstly, the systems and formats of research data can differ widely: From being stored in different types of databases (relational, key-value, graph or document databases) to files in different formats (such as the MRT format for BGP data, or zone files for DNS data). Retrieving and representing such data is difficult. Furthermore, data is often distributed over the network and not centrally accessible on a single machine. Finding a method to allow easy access to and representation of such heterogeneous data is a key challenge in this project.

Another difficulty is the huge amount of data: A scan of the full IPv4 address space can result in billions of data items, and multiple scans quickly increase the total data available. Here, a challenge is to find a space-efficient storage method that still allows access to the full original experiment data in an efficient way.

Additionally, access to the data is not a big advantage if the semantics of the data are not clear. If the original researcher has not made his exact methods for accessing and selecting the data public, it will be difficult to use the data for meaningful experiments, even if it can be accessed easily enough. Thus, a secondary objective of the Empirator system will be the addition of small amounts of metadata to the original data source to facilitate data reuse without contacting the original experimenter.

The Empirator system we propose will help alleviate these concerns by providing a platform for central data storage and analysis. In the following chapters 2.1 and 2.2, we will provide some background information for the reader, as well as present work related to this endeavor. In chapters 3 and 4, we will describe design and implementation of the system. In chapters 5, the performance of the implemented system will be evaluated. In chapter 6, we will describe avenues for further research, as well as summarize the results of this work.

Chapter 2

Background & Related Work

2.1 Background

2.1.1 Internet-Wide Scanning

The internet has grown to such an extent that the number of available IPv4 addresses has almost been exhausted. This means that the internet now consists of more than 3 billion hosts.

In order to collect reliable data on the internet's structure and composition, internet-wide scans can be used. The emergence of new tools for mass scanning like ZMap [1] or masscan[1] and the wide availability of high bandwidth networking has made such scans much easier and faster.

An internet wide scan consists of enumerating all non-reserved IP addresses, and sending some sort of probe to each, or a subset, of all available addresses. Probes can be simple (like ICMP echo requests), or more complex (like a complete SSL handshake). The target and response to the probe are then logged.

Scanning the internet in this way results in large amounts of data, depending on the information logged, and scanning may take a long time, depending on the bandwidth used to scan.

The resulting datasets have a variety of uses, particularly in the field of security research. Prominent examples include analysis of the SSH protocol [2], and evaluation of TLS security [3].

However, the results of such scans must be examined carefully. Factors such as scan location and scan duration strongly influence the information that can be gained from

[1] `https://github.com/robertdavidgraham/masscan`

a scan. For example, network segments can be unreachable for a certain time, and reappear during the scan, leading to partial data for that network.

Additionally, network traffic is often modified by a number of network hosts, depending on network origin, location of the sender or traffic. This might lead to scan results being modified depending on such factors. Examples for these kinds of traffic modification include firewalls, load balancers and network address translation.

Nonetheless, important information can be gained from analyzing such scans.

2.1.2 The BGP protocol

The border gateway protocol (BGP) is used to distribute routing information across autonomous systems. Therefore, BGP forms the basis of internet-wide routing decisions. The current version of BGP is described in RFC 4271 [4].

To participate in BGP, each protocol speaker is configured to exchange information with one or more peers. Once connected, these peers exchange routing information over the network. This collected routing information (also called RIB, “routing information base”), combined with locally configured policies, is then used by the participating routers to select a route to include in the local routing table.

Routing information can be announced (prompting all peers to store this routing information in their Loc-RIB) or withdrawn (causing peers to remove it from their RIB, and potentially from their routing table). Routing information always pertains to prefixes of IP addresses.

BGP packets can contain either “management” level functions like Open, RouteRefresh or Notification, which will not be used in this work, or connection information in an Update message. Each update message contains a number of prefixes that were withdrawn by the peer (meaning the prefixes are no longer reachable via that peer) and a number of announced prefixes (meaning the prefixes are reachable via that peer), as well as a number of attributes applying to the announced prefixes. These attributes can be required or optional. The three required attributes of an update message are the following:

Origin How this routing information was created

AS path The list of AS through which this information has passed

Next hop The next hop of a path to reach the advertised prefix

Additionally, BGP packets often carry a number of additional, but optional attributes for influencing routing decisions, such as the multi-exit discriminator or the local-preference field used for routing decisions.

Since BGP forms the basis of a large number of routing decisions in the internet, analysis of the protocol is very useful for understanding the structure of the internet.

2.1.3 TLS and X.509

TLS and its predecessor SSL are cryptographic protocols using asymmetric cryptography to ensure confidentiality, integrity and authenticity of messages. In particular, TLS and SSL are widely deployed in combination with HTTP as HTTPS for secure web communication. TLS and SSL both rely on the X.509 standard, which defines standards for public key infrastructure and public key certificates. TLS allows the exchange of arbitrary application data over an encrypted channel, and defines a wide variety of primitives to set up this channel.

TLS 1.2 supports 19 key exchange methods, 9 encryption cipher/encryption mode variants and 4 data integrity functions. While this allows users to choose encryption settings that match their requirements, this high number of configuration possibilities also increases the potential for misconfiguration, thereby making the connection insecure. While high modularity makes it possible to quickly migrate from insecure ciphers to secure ones, this migration often happens very slowly. Thus, it is interesting to see how deployments of X.509 are configured, and how the use of secure primitive combinations changes over time.

The X.509 public-key infrastructure (PKI) used by TLS is another subject that lends itself to study. The X.509 PKI is described in RFC 5280 [5]. Its basic structure is as follows: Participants identify themselves via a certificate, which contains the public key of that participant as well as identifying information (such as name, or website URL). In order to prove the correctness of this information, the certificate is signed by a third party's private key. In order to verify this signature, another certificate is needed, and so on. This certificate chain ends at some certificate that is implicitly trusted. This certificate is called *root certificate*. These root certificates are issued by certificate authorities (CAs), who check whether the information in the certificates they sign is correct.

While this is theoretically sound, CAs often fail to correctly verify certificates, or issue certificates that can be misused. In recent times, several incidents with misconfigured CAs or wrongly issued certificates have occurred. It is thus interesting to analyze which CAs have issued certificates for whom, and whether these certificates are used and issued correctly.

2.2 Related Work

Unifying the storage and analysis of network research data and allowing combination of disparate data is not a novel idea. [6] proposes a database scheme for sharing such measurement data called SIMR. It proposes a more detailed and specific database schema than this work, but has not been implemented.

Examples for implementation of a scheme inspired by SIMR include the Internet Measurement Data Catalog (IMDC) [7][2] and the IST-MOME database [8] [9]. These system are designed to allow replayability and reproducibility of network research by making the raw experiment data available to the scientific public, and allowing easy integration of new data sources and measurement types. Both the IMDC and the MOME database share some characteristics with this project. However, both systems are centrally deployed, and do not offer programmatic access to parts of the measurement data — only whole datasets can be retrieved. Additionally, the scope of both the IMDC and IST-MOME is much bigger than that of this work.

Integrated measurement systems like mPlane [10], PerfSONAR [11] or InterMON [12] allow centralized analysis of network data as well, but their primary aim is the integrated creation of more measurement data, and usage of that data for locating and eliminating network problems, which is not a primary concern of the Empirator system.

In recent years, research centering on full network scans has become more common, and it is expected a majority of the data in network measurement management systems system will consist of such scans. Some recent scanning efforts are described in [13], [14] and [15]. The results of these scans can be used for a variety of purposes, such as evaluating performance characteristics or improving network models. Additionally, organizations like scans.io [16] periodically make measurement data publicly available.

One important application of full network scans is the analysis of security properties of the internet. Examples for this kind of analysis include [17], [2], [3], [18] and [19].

[2] `http://imdc.datcat.org/`

Chapter 3

System Design

The Empirator system is designed to allow access to heterogeneous network research data in a homogeneous way.

Some motivation for design and implementation have been outlined in chapter 1. However, a more thorough investigation of the goals of this project will help with design. We will introduce a number of use cases to illustrate the features of Empirator.

The Empirator system can be used for improving existing research: While examining scan data of some sort (say a SSH key scan) using the Empirator system, a researcher notices that several addresses in some subnetworks share an implementation flaw (perhaps weak keys that were generated with insufficient entropy). By navigating to the Empirator web interface, and entering some of these addresses, the researcher notices that reverse DNS data is available for the time period during which the data was collected. He writes a small script that collects reverse DNS data for all networks using the Empirator system, and notices that most of the addresses' DNS entries were registered only a short time before the scan. Thus, one reason for the experiment results could be some default configuration in the hosts that will be corrected soon. Here, the Empirator system, by combining research information, allows more precise research.

Another use of the Empirator system is the creation of new research by correlating already existing data. While idly browsing the Empirator web interface, a researcher notices that both SSL scan information and BGP data is contained in the interface. The researcher writes a script retrieving both SSL certificates for a certain time period as well as BGP announcements for that period. By correlating BGP changes and SSL certificates, the researcher notices a correlation between SSL certificates and unrouted prefixes. This gives the researcher an idea for an experiment, to research where this correlation does not hold. Again, this experiment can be executed using the Empirator system.

In addition to support the use cases above, Empirator tries to keep complexity of the system on a manageable level. While projects such as IST-MOME [9] have more features than planned for Empirator, deployment and development of such large and complicated systems often fails. Additionally, introduction of new data is often complicated, requiring data conversion and rigid adherence to the predefined structure of the system.

In contrast, the Empirator system aims for the least necessary complexity, while supporting the use cases described in the previous paragraphs. This does not only influence the Empirator code base, but also the requirements for implementing new data sources: Simplicity, easy extendability and convenience for researchers are additional aims of the Empirator system.

The design decisions taken to fulfill these aims will be described in the next chapters.

3.1 Data Representation

Because the data contained in the Empirator system is heterogeneous, a structured format for representation of this data is hard to devise. While it is conceivable to design a structured format for representing the full range of data produced by different research projects, the extensibility of the system would suffer from such a rigid data format. Being unable to add new data sources to the system without modifying the basic data representation would make extending the system very difficult, and would thus not be desirable.

In order to design a data representation format, some assumptions have been made about the data to be stored in the system: We assume most network research data that falls under the umbrella of the Empirator system will have a common structure. Most network research experiments consist of a mapping from some target or *object* to some data returned by the experiment for that object. Possible targets for such scans include IP addresses or AS numbers, while possible data includes measurements such as X.509 certificate fields, reverse DNS entries or SSH banners. In the rest of this text, the subjects of an experiment will consistently be referred to as *object*, while the general term *data* refers to the results of an experiment for a certain object.

It is evident that the number of objects is limited — there is only a small number of identifiers that can be targeted by network research experiments. Among these, three were selected for inclusion in the Empirator system: *IP networks*, *AS numbers* and *DNS names*. In the opinion of the author, these objects cover a large proportion of network research, especially those that result in large datasets.

In contrast, the data returned by a scan is difficult to classify. One can quickly think of a large number of different experiments that return totally different types of data: SSL scans map IP addresses or DNS names to certificate information, FTP scans map IP addresses to FTP banners, port scans map IP addresses to open ports or operating system configuration.

Object	Timestamp	Data items
8.8.8.1/32	2015-04-15 13:30	cert. issuer: cacert.org, valid: false, …
9.9.9.1/32	2015-04-16 13:32	cert. issuer: Thawte, valid: true, …

(a) SSL scan: Objects are IP addresses

Object	Timestamp	Data items
www.google.de	2015-05-15 16:01	A: 216.58.211.3, AAAA: 2a00:1450:4016:803::100f, …
net.in.tum.de	2015-05-16 17:22	A: 131.159.15.49, AAAA: 2001:4ca0:2001:13:250:56ff:fe9d:955, …

(b) DNS scan: Objects are DNS names

Table 3.1: Data representation example

However, every result of a networking experiment has a single property in common: the time of its creation (also referred to as "time of observation"). Every measurement occurred at a certain time, and this timestamp can be immensely useful for analysis and combination of scan data.

Following from this, a tabular data format will be selected for the Empirator system, where each measurement point has two fixed properties: the object it refers to (IP network or address, AS numbers or DNS names) and the time of its measurement. The actual results of the experiment will be represented in a flexible tabular format. The data is represented in string format. Choosing a simple string serialization allows the representation of almost arbitrary data, not restricting the domain of the Empirator system. Additionally, no serialization process will be proscribed by the Empirator data. However, by looking at existing data, researchers will be able to standardize on a "core" of well-known string serialization formats (such as for IP addresses, or time stamps).

An example data representation of two network scanning experiments is shown in figure 3.1.

3.2 Data Storage

All data in the system stems from external sources — prior experiments or public research data. The amount of data such experiments generate is potentially very large, compounded by the fact that keeping historical data is often useful, which again increases storage requirements. For example, a single IPv4-wide SSL handshake scan [16] weighs in at about 100 GB[1] as compressed data, or about 200 GB without compression. Similarly, a single BGP routing table dump in uncompressed MRT format[2] takes about 500 MB of storage, and multiple such files are produced every day.

Uncompressed, and with daily scans from several sources, the total amount of data that would be stored in the system would be so large that only storing the data in the system would be a challenge, never mind devising a system for querying and retrieving this data. Moving all data into the system itself would mean either keeping duplicate data or removing the data from the original experiment. Keeping duplicate data would consume a disproportionate amount of storage, while removing the data from its original source would mean removing the data from the original author's grasp. This would be significant hurdle for the adoption of the Empirator system.

A better solution is keeping the data in the original storage system, and transferring it to the user of the system on demand. While this will decrease performance, it significantly lessens the burden on the data producers, as well as the administrators of the Empirator system.

3.3 Data Access

The experiment data that will be processed by the system is stored in different data storage systems with different access methods, like relational databases and plain text files, among others. Building a single central component to allow access to all of these systems (and keeping access methods extensible) would go beyond the scope of this work.

Additionally, simply accessing the data on a mechanical level would not be enough — the *semantics* of the data would be entirely unclear to the system and its users without extensive configuration by the original researcher. Again, designing a configuration system to describe the semantics of arbitrary data would fall beyond the scope of this work.

Instead, each data repository will be accessed via its own custom access component, the *adapter*. Each adapter is adapted to the repository's data storage, and contains knowledge about the semantics and relation of the data items contained in the repository. While this leads to a higher implementation effort by the original researcher, it does not in any way limit the addition of new data sources.

Having the full power of a programming language available allows any processing necessary for data access to be implemented in the adapter. Additionally, infrastructure

[1] `https://scans.io/series/443-https-tls-full_ipv4`

[2] `http://data.ris.ripe.net/rrc00/2015.04/bview.20150422.0800.gz`

that frees the implementer of these access components from the administrative burdens of communicating with the system and simplifies accessing the experiment data will be provided. An adapter together with the data it accesses will be known as *data source* (“source” for short) in the remainder of this text.

In order to allow users of the Empirator system to make sense of the data made available by the adapters, some metadata will be attached to each adapter. This metadata will describe the available data in a human-readable format. This metadata is only meant to help human operators of the system interpret the available data and describe possible values, and is not supposed to be machine-readable.

A machine-readable metadata format for describing data would allow the Empirator system to automatically convert data into standard formats, or check the validity of returned data. However, the author argues that using such a strict format would greatly reduce the flexibility of the system.

3.4 Data Selection and Retrieval

In order to fulfill its stated functions, the Empirator system must provide functionality to retrieve a subset of available data. This allows clients to only process data that they need, and allows targeted research on the data stored in the Empirator system.

Owing to the flexibility of the system, as well as the decentralized nature of its data storage, only a limited number of selection methods for data can be supported. In particular, the system will only be able to support selection of data by the shared common properties of data items — the object and the timestamp of data. Filtering data by other properties is not possible, because that data is not available to the central server.

While this might seem limiting in comparison to filtering possibilities in the original data storage system (be it a database or binary files), where the full data of the experiment is available, the retriever of the data can always do any specialized filtering locally, after retrieving the data from the Empirator system. Relying on the system to provide a selection and filtering interface for all possible data sources would lead to high complexity and an abundance of corner cases that would make using such a system difficult and error-prone.

Supported selection and filtering operations should include such actions as selecting all data created in a certain time range, all data regarding a specific object or a set of related objects (such as all data containing entries regarding a specific network and all its subnetworks, or all DNS entries ending in *google.com*) and any combination of these properties.

Combining requirements for decentralized data storage and data selection means that data selection will either be handled decentrally, by every adapter connected to the system, or by a central component. Handling data selection in a decentralized fashion would improve performance for retrieval of data from only a single source, but would require contacting every data source connected to the system and interrogating it for available data. Also, this would lead to a duplication of efforts: Each adapter would have to implement some sort of indexing solution optimized for the type of data it supports. Requiring the implementer of an adapter to implement an index would be a significant effort, and thus impact the user-friendliness of the system.

A centralized selection system, on the other hand, would simplify the implementation of adapters and allow quicker filtering and selection of available data without contacting attached adapters for available data.

Implementing a centralized filtering system requires the introduction of two activities during the lifetime of the system: *Indexing* and *Retrieval*.

During the indexing phase, metadata is sent from the adapters to a central system, called the *index*, where it is stored. When data is requested from the system, matching metadata is retrieved from the index and used to retrieve the actual data from the connected adapters. We argue that this two-stage approach will allow efficient storage and retrieval of data, as well as simplify implementation of additional adapters.

3.5 Filters

Not all interesting network data is available in the form of large data dumps. A lot of data is primarily available through online services, such as WHOIS data[3] or historic DNS data[4]. While implementing a proper scan infrastructure consisting of a download program and an adapter would be the cleanest way to integrate such data into the Empirator system, allowing external processes to augment data passed through the system is simpler way of providing access to additional data sources.

This will be done in the Empirator system by a component called *filters* in the rest of this document. This name was chosen purposefully, for the similarity to the Unix concept of a filter: A filter takes some input, processes it in some manner, and then outputs the result again.

In the case of Empirator, a filter receives as input the data returned by some query. The filter then executes some processing on the resulting data — either augmenting it with more information, or reducing the data for for some next step. Then, the filter sends

[3]Most easily available through the `whois` command

[4]E.g. `https://dnshistory.org/` or `http://whoisrequest.org/history/`

the data back to the Empirator system to be sent to the client. Clients can choose which filters to apply to a query, and can pass arguments to those filters.

Implementing this feature allows easy access to online data sources and additional processing that is not supported by the Empirator system directly.

3.6 Toolchaining

Most network research experiments follow a similar process for their execution: Data is generated by a scan, preprocessed by some program, and finally saved in an easily accessible format for analysis. Often, such experiments are executed periodically, to generate more data and make it possible to discern historical trends or correlations. It is advantageous to automate the collection of such data. Often, some sort of automation is already available for data collection.

The Empirator system aims to allow easy integration of such automation into the system. Thus, Empirator will provide scan implementers with some kind of *toolchaining* support: Starting scans and integrating the newly arrived data into the Empirator should be possible with as little effort as possible.

The capabilities planned for toolchaining are support for starting updates and some mechanism to notify the Empirator system of newly arrived data, enabling quick integration of new data into the system.

Chapter 4

System Implementation

Guided by the design considerations in the previous chapter, a system was implemented that fulfills the listed requirements. In the next chapter, we will shortly describe the technology used in the implementation of the Empirator system. Then, in chapter 4.2, the components making up the Empirator system will be described in detail: The index, adapters and the user interface of the system will be described in detail. Finally, in chapter 4.3 the communication protocol used in the Empirator system will be described.

4.1 Core Technologies

The core implementation language for the Empirator system is the Go language[1]. Go is a statically typed, compiled, garbage collected language that is designed for building high-performance concurrent services. The Go language has several desirable features that ease implementation of the Empirator system: Concurrency support, static typing and an extensive standard library. External Go dependencies of this project are the `pq` library for PostgreSQL database access[2] and the `zmq4` library for ZeroMQ support[3].

Other languages considered for the Empirator core implementation, but ultimately rejected, were Java, Python and C++, due to performance and complexity concerns.

An additional language used in the system is Python[4], an interpreted high-level language well suited for exploratory programming. The libraries `requests`[5], `ipaddress`[6], `pytz`[7] and `tzlocal`[8] for web interface access, IP parsing, and timezone calculations,

[1]See `http://golang.org/` and [20]
[2]See `http://github.com/lib/pq`
[3]See `http://github.com/pebbe/zmq4`
[4]see `https://www.python.org/`
[5]See `http://python-requests.org`
[6]See `http://pypi.python.org/pypi/py2-ipaddress/`
[7]See `http://pytz.sourceforge.net/`
[8]See `http://pypi.python.org/pypi/tzlocal`

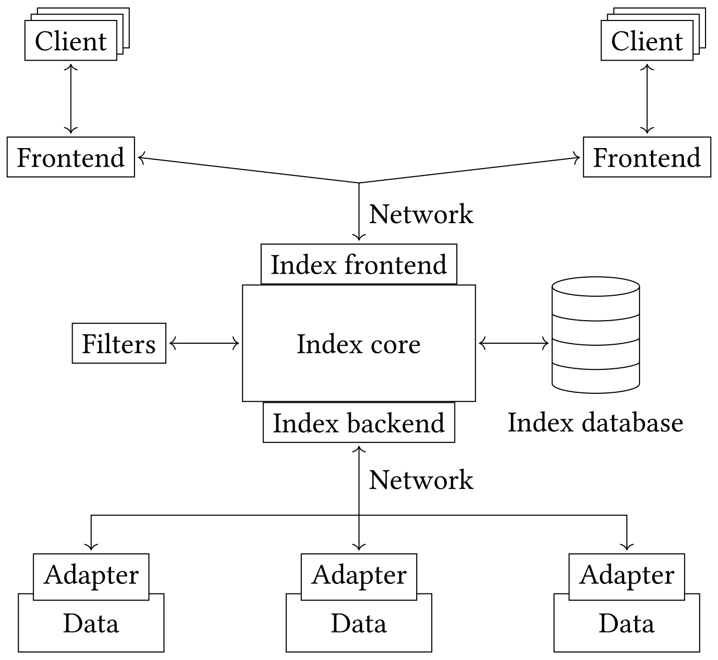


Figure 4.1: System architecture

respectively.

For communication with adapters over the network, the *ZeroMQ*[9] (or ØMQ) messaging library was chosen. ZeroMQ is a high-performance, asynchronous, brokerless messaging library consisting of a thin abstraction layer over a socket interface. Reasons for choosing ZeroMQ include low complexity and high performance. Messaging using ZeroMQ is simpler than using a socket directly, and significantly less complex than choosing a higher-level communication system, such as a broker-based messaging system like RabbitMQ or XML-based systems such as SOAP.

Additionally, for communicating with clients and internal data representation, a JSON representation for data was chosen. Reasons for this were the low effort for serializing and deserializing JSON objects into normal objects, as well as good support in the standard libraries of the chosen implementation languages.

4.2 System components

The main components of the Empirator system are the following: The *Index*, consisting of a core, index front- and backend, *adapters* and user interfaces, also called *frontends*. Each of these components and the relations between them are depicted in figure 4.1.

[9] see `http://zeromq.org/`

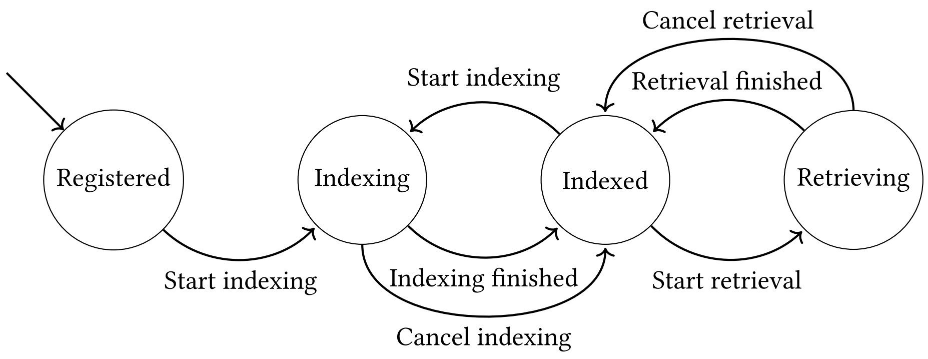


Figure 4.2: Adapter states

The index is the central component of the system. Filters, the index database and the index core will be described in section 4.2.1. The index maintains a connection to each adapter, mediates data retrieval and post-processing, and provides a way to access that information for user interfaces.

Adapters allow accessing data sources, supporting both indexing and retrieval. Their general structure and functionality will be described in chapter 4.2.2. Additionally, two available adapters and some of their implementation details will be described.

Finally, user interfaces allow users to access the data contained within the Empirator system. The implementation of user interfaces and two available user interfaces will be described in chapter 4.2.3.

Adapters and server are connected over the network.

4.2.1 Index

The index is the core component of the Empirator system. Guided by the considerations in chapter 3, the index orchestrates the selection and retrieval of data from data sources.

The index has three major functions: Maintaining a connection with clients and adapters, storing indexing data, and making that data available for retrieval. These responsibilities, as they apply to adapters, will be described in more detail in the next chapters, while frontend functionality provided by the index will be described in chapter 4.2.3.

An adapter connected to the Empirator system is always in one of four different states: Registered, Indexing, Indexed or Retrieving. These states and the available transitions between them are illustrated in figure 4.2.

When initially connecting, an adapter is in the *Registered* state. No indexing has taken place on the adapter yet, and no data can be retrieved. When indexing on an adapter is started, it transitions into the *Indexing* state, during which indexing data is sent to the server.

When an adapter has finished indexing, the state of the adapter transitions to the *Indexed* state. This is the “default” state for connected adapters, and when reconnected, the adapter will be transitioned to the Indexed state directly, without requiring a new indexing.

Finally, when data is requested from an adapter, the adapter transitions to the *Retrieving* state. In the Retrieving state, the adapter sends data items to the server. After all data has been sent by the adapter, the adapter transitions back to the Indexed state, and retrieving or indexing can be started again. There is no “Toolchaining” state in this diagram, because the toolchaining status of adapters is not kept on the server. Toolchaining is implemented entirely on the adapter side, and thus the server does not need to keep track of the current toolchaining state.

As can be seen in figure 4.2, an adapter is either in the indexing or retrieving state, never both. This entails that no data can be retrieved from the adapter while it is being indexed. This limitation was chosen because it makes adapter behavior much easier to reason about, and significantly decreases implementation complexity of both adapters and server.

Both indexing and retrieval can be canceled: If a cancellation message is received by the adapter during indexing or retrieval, all work is aborted and the adapter returns to the *Indexed* state.

All connected adapters (regardless of state) are contacted regularly by the server to test whether they are still connected. If no reply is received for a certain time, the adapter is marked as disconnected, and indexing or retrieval is canceled.

4.2.1.1 Adapter Implementation

Adapters need to register with the server prior to any system actions taking place. This is done by sending a registration message to the server. This registration message contains the name of the adapter, the supported object types of the adapter (a subset of IP networks, DNS names and AS numbers) and a human-readable description of the data provided by the adapter (the metadata mentioned in chapter 3.1). This metadata consists of a list of string pairs, describing the mnemonic names of the adapter’s data items, as well as a plaintext description of their content. For an example description, as could be sent by an adapter allowing access to port scan data, see table 4.1.

Adapters need not be preconfigured: Adapters, once registered, are automatically stored and reconfigured when the index service is restarted. This eases connection of new adapters. An implementer must simply code an adapter with the correct implementation,

Name	Description
name	This column contains the DNS name of the scanned host
duration	This column contains the time it took to completely scan the host
ports	The list of open ports for this host, separated by commas.
os	The operating system for this host.

Table 4.1: Example description

configure the index address and start the adapter. The adapter will register with the index, and be available for all future system actions.

4.2.1.2 Indexing

During indexing, information about available data is retrieved from adapters and stored in the index database. This information consists of the "common properties" described in chapter 3.1 (object and timestamp) and a *key*.

While the object describes the target this data is associated with, and the timestamp describes the date of observation, the key is an adapter-specific value that is provided to the adapter to retrieve the associated data item. Since this key is totally opaque to the server, it is handled as a binary value. The key for an adapter for reading plain-text files, for example, could be generated from a combination of filename and line number describing the location of a set of data.

The combination of adapter name, object, timestamp and key will be termed *index item* during the rest of this document.

Storing only a key and not full data allows efficient data retrieval from heterogeneous sources such as databases, where a table's primary key can be used as the key, or plaintext files, where line numbers or byte positions can be used, and reduces the amount of data stored in the central index database. Allowing the adapter to generate the key allows storing arbitrary location information for data, which allows adapters to be implemented for accessing data stored in all kinds of forms.

Indexing a single adapter consists of the following steps:

- Starting indexing on adapter
- Receive index items from adapter
- Save index items to database

- Wait for completion indication from client

This process is pipelined: While items are being inserted into the database, more items are already being read from the socket, and more items are being sent by the client.

Indexing can be run more than once: Whenever new data is available, an indexing run must complete before the new data is accessible in the system. To prevent multiple index entries referring to the same data, a timestamp is used when starting indexing on a client. Only items whose timestamp is later than the specified date should be sent by the client.

An adapter can choose to associate several time/object tuples with a single key (*key batching*). When data is requested from the index, the index selects all keys for matching objects from the database. When the full data is retrieved from the adapter, it is post-processed to remove any data not matching the request. This allows every adapter to assign an arbitrary number of items to each key, allowing it to exploit relations between data items (such as being saved on adjacent lines in a data file) and reduce the amount of key data contained in the index database.

At any time during the indexing process, the adapter can send an error message to abort the indexing. All items received so far are discarded, and the adapter is reset to its regular state. If the adapter is disconnected during indexing, the indexing process is aborted as well, and no data is written to the database.

The received index items are stored in the index database, which is described in the next chapter.

4.2.1.3 The Index Database

The index database stores the data retrieved from adapters during indexing. It is implemented using the PostgreSQL[10] database. Reasons for the selection of a relational database system were familiarity with the database system, and the good fit of the processed data with the relational paradigm.

A diagram of the database layout can be found in figure 4.3. The “source” table stores information about connected adapters and their metadata, while the “items” table stores index items.

In the source table, the index stores the name, the supported types, the description of a source and when the source was last updated. When the index is started, this data is read from the database. When an adapter connects to the system, the stored data (if present) for that adapter is retrieved and associated with the newly connected adapter. This way, information such as the last indexed timestamp can be stored persistently.

[10] `http://www.postgresql.org/`

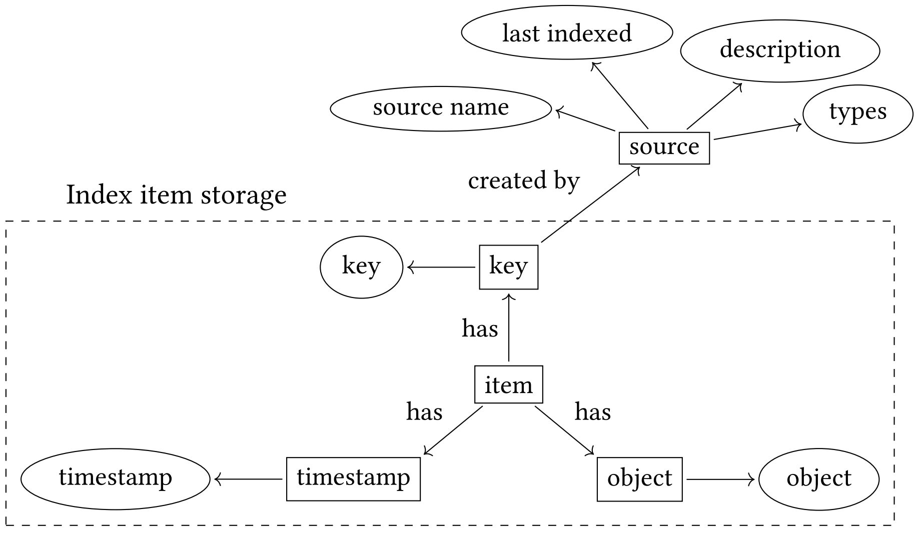


Figure 4.3: Index database layout.
"Item", "key" and "object" tables are duplicated for each supported object type (IP, AS and DNS).

An index item is stored as an association between timestamp, object and key. The value of each of these properties is stored in a different table, which was done to allow "key batching" as described in the previous chapter.

When a client requests data, the index database is used to collect all keys associated to data matching that request, and the keys are used to collect the data from all adapters.

New index items are added to the index database during indexing, and source information is updated whenever indexing completes successfully, or when the index is shut down.

4.2.1.4 Filters

The design of filters in the Empirator system is described in chapter 3.5. Filters are implemented in the Empirator system in the following way:

Each filter consists of a configuration file describing the filter, and a filter binary doing the actual filtering. The filter config file is a JSON file that needs to contain the following settings:

BinaryPath The path to the filter binary.

WorkDirectory Which work directory to use while executing the filter.

Description A human-readable description of the filter's effects and usage. Should include a description of the filter argument format.

Object	Adapter	Timestamp	Data items
www.google.de	Adp1	2015-05-15 16:01	A: 216.58.211.3, AAAA: 2a00:1450:4016:803::100f, …
net.in.tum.de	Adp1	2015-05-16 17:22	A: 131.159.15.49, AAAA: 2001:4ca0:2001:13:250:56ff:fe9d:955, …

(a) Data before filtering.

Object	Adapter	Timestamp	Data items
www.google.de	Adp1	2015-05-15 16:01	A: 216.58.211.3, AAAA: 2a00:1450:4016:803::100f, A-rdns: muc03s14-in-f3.1e100. net.,…
net.in.tum.de	Adp1	2015-05-16 17:22	A: 131.159.15.49, AAAA: 2001:4ca0:2001:13:250:56ff:fe9d:955, A-rdns: typo3.net.in.tum.de.,…

(b) Data after filtering with the reverse-dns filter. The filter was passed "Adp1:A" as argument.

Table 4.2: Data filtering example

When run, the index reads all ".conf" files in a configurable subdirectory, and lists those filters as available by the user.

When data is filtered, the filter binary is executed, with the command line arguments selected by the user, and data is piped into the filter via its standard input. If the filter encounters an error making it unable to process the data, it should exit with a non-zero exit status, and print an error to standard error. This error will be shown to the client. If no error occurred, the filter should send the data to its standard output.

A filter can modify input data in any way, appending new data columns to each data item, adding entirely new objects or adding data items to certain objects. Two filters are currently implemented: "as-name" calls `whois AS<AS number>` for a configurable number of data item columns, while "reverse-dns" adds reverse DNS entries for specified columns to the data item. Both filters can be configured using arguments of the following form: `<adapter name>:<column_name>[,...]`, which means that the column with name `<column_name>` is processed by the filter for every data item from the adapter `<adapter name>`. An example for data filtering using the reverse DNS filter can be seen in table 4.2. The top table shows the data before filtering, while the bottom table shows data after filtering. The filtered data now contains a reverse DNS entry for every A entry contained in the data previously.

Note that a filter always receives the full result set of a query. This means that running filters on large queries may lead to excessive memory use by the index.

4.2.1.5 Queries

Guided by the considerations in chapter 3.4, a method of querying was designed for the Empirator system. A query is used to select the data a client wants to retrieve. Each query consists of a "selector", an object type and optional arguments to the query. The selector describes which data to select from the system, the object type describes the types of objects returned by the query, and arguments modify the query itself (used for limited and filtered queries, described later).

The Empirator system supports three different types of selectors:

Object-specific For selecting data based on the object. Examples include selecting all data with a selected IP address, or all data where the DNS name matches "*.google.com".

Object-agnostic For selecting data based on object-independent properties. Examples include selecting data based on timestamp, or based on the adapter name which stores that data.

Boolean For combining queries from the previous categories. Examples are AND, OR and NOT.

A query consists of an arbitrary combination of these selectors, where only object-specific selectors with the same object type as the overall query can be used.

For a full list of available selectors, see table 4.3. When data is requested from the index, the query contained in that request is translated to SQL and passed to the index database. Data is requested from the client, and the retrieved data is passed through the query again. This removes data potentially added by key batching. Then, the data is passed back to the user.

The Empirator system offers different types of queries to fulfill different requirements. The available query types are:

Standard Retrieve all data matching the provided selector.

Limited Retrieve a limited number of data items from the index. Limit is given as argument.

Filtered Retrieve all data matching the provided selector, and apply filters. Filter names and arguments are given as query argument.

Count Count the number of items in the database that match the specified selector. No retrieval takes place.

Note that Count query might have confusing results due to clients potentially returning more than one data item for an entry in the index database. The results of a count query

Name	Arguments	Description
`IP_CONTAINS`	CIDR network	Select all data items having an object that is a subnet of the argument.
`IP_IS_CONTAINED`	CIDR network	Select all data items having an object which the argument is a subnet of.
`IP_EXACT`	CIDR network	Select all data items having the same object as the argument.
`AS_EXACT`	AS number	Select all data items having the same object as the argument.
`DNS_EXACT`	DNS name	Select all data items having the same object as the argument.
`DNS_LIKE`	SQL "like" expression	Select all data items having an object that matches the "like" expression.
`TIMESTAMP_AFTER`	timestamp	Select all data items that have a timestamp after the argument.
`TIMESTAMP_BEFORE`	timestamp	Select all data items that have a timestamp before the argument
`SOURCE_EXACT`	source name	Select all data items that were created by the adapter with the name of the argument.
`AND`	Two subqueries	Select all data items that match both subqueries.
`OR`	Two subqueries	Select all data items that match either of both subqueries.
`NOT`	One subquery	Select all data items that do not match the subquery.

Table 4.3: Available data selectors

mean that a standard query with the same arguments will always return at least as many items as the result of the count query.

The availability of these selectors goes a long way towards alleviating the reduced selection possibilities of the Empirator system caused by decentralized data storage.

4.2.1.6 Retrieval

Retrieval works similarly to indexing, but across several adapters, since data from multiple adapters might be received.

When data is requested by a client, all index items in the index database matching the request are selected, and the keys are dispatched to the correct adapter. In response, the adapter sends over the actual data. The index then collects and post-processes that data. Post-processing includes merging data from different sources and filtering out

Object	Adapter	Timestamp	Data
8.8.8.0/24	Adp. 1	1970-01-01T12:30:00Z	Origin: EGP, MED: 3, …
		1970-01-01T12:35:00Z	Origin: IGP, MED: 100, …
		1980-01-01T12:30:00Z	Origin: IGP, MED: None, …
	Adp. 3	1980-01-02T12:32:00Z	OS: unix, Ports: 21,22,23,…
8.8.9.9/32	Adp. 2	1980-01-01T12:30:00Z	Error: None, Traceroute: …
		1970-01-01T12:40:00Z	Error: None, Traceroute: …
		1970-01-02T12:50:00Z	Error: None, Traceroute: …
	Adp.3	1980-01-01T12:31:01Z	OS: unix, Ports: 22,80,8080,…

Table 4.4: Data representation in the Empirator system

non-matching items introduced by key batching. If filtering is requested by the client, the data is passed through all filters that have been selected by the client, and then made available to the client.

Similarly to indexing, retrieval is pipelined: While the adapter is sending data, the server is already post-processing the data stream and making it ready for retrieval by the client.

Retrieved data is handled in a format that will be termed *data item* in the rest of this document, analogously to “index item”. A data item consists of an object, a timestamp and a list of strings containing the data. We did not proscribe standard ways to encode data in string format, since this would again reduce flexibility of the system. We argue that newly implemented adapters will serialize data in a similar format to existing adapters, thus leading to a set of standard serialization formats (such as serializing IP addresses with a netmask, or using RFC3339 format as the standard timestamp format).

A collection of data items is organized as shown in table 4.4: First by the object of the data item, then by the source the data item came from. Each line in the table displays a single data item.

4.2.2 Adapters

While the index is the core of the Empirator system, adapters serve to make it useful. An adapter is the custom code required to access data and connect it to the Empirator system.

For fully participating in the Empirator system, an adapter must be able to do the following things:

1. Maintain a connection to the server
2. Register with the index

```go
func (a *MyAdapter) GetIndexItems(last_indexed time.Time,
        output chan<- *objects.IndexWithError,
        cancel <-chan interface{}) {
        //Implement generation of index item here
}

func (a *MyAdapter) GetDataItems(objtype objects.ObjectType,
        keys []Key,
        output chan<- *objects.DataWithError,
        cancel <-chan interface{}) {
        //Implement retrieval of data items here
}
func main() {
        adapter := MyAdapter{}
        c := client.NewServerClient(&adapter, "serveraddress", "")
        c.SetSourceInfo("AdapterName",
                []objects.DescField{
                        objects.DescField{"Field 1", "Description 1"},
                        objects.DescField{"Field 2", "Description 2"}},
                objects.OBJECTSET_IP)
        c.Loop()
}
```

Listing 1: Sample adapter code

3. Send index items
4. Receive keys and returning data

Optionally, an adapter can support toolchaining as described in chapter 3.6.

All existing adapters are programmed in Go. For implementing new adapters in the Go language, extensive support code is provided that greatly reduces the burden of implementing new adapters. Items 1 and 2 are taken care of by the adapter support code: Networking (i.e. responding to pings in a timely fashion, as described in chapter 4.3) is entirely automatic, and network registration consists of a single function call into the provided support code. An example adapter skeleton is shown in listing 1.

This code (with added imports and error handling) is already a fully functional, albeit quite useless, adapter. If indexing is requested, success is immediately signaled to the index without sending any keys. If keys are requested, no data is sent in reply. The `Loop()` function takes care of keeping the adapter connected to the server, item sending and receiving as well as cancellation.

The only difficult implementation work needing to be done by the user are the `GetIndexItems` and `GetDataItems` functions.

The GetIndexItems function is used for generating index items. The last_indexed argument tells the adapter which index items to send to the server. All index items that have been observed later than the last_indexed timestamp must be sent to the server. The output argument is the sink for index items. The channel is automatically closed when the function returns, or an error is sent by the adapter code. The cancel channel is used for canceling a running indexing progress. When the cancel channel is closed, the adapter implementation should return as soon as possible and clean up all open resources.

The interface of the GetDataItems function is very similar. The objtype argument identifies the type of object to retrieve, while the keys argument contains the adapter-specific key needed to retrieve the data. Again, data should be sent to the output channel, which is automatically closed when the function returns or an error occurs. The cancel channel is used identically to the GetIndexItems function.

For implementing these functions, the implementer has to make a decision about the keying scheme used in the adapter: A key needs to uniquely identify a data item, and should be as compact as possible to reduce the size of the indexing database. Another consideration should be providing efficient access to the data source. This works differently for every storage system, and thus, no general guidelines can be given here. For examples of keys, see chapters 4.2.2.1 and 4.2.2.2.

```go
// Creating a key
var schemaname string = "test schema"
var host string = "test host"
var port uint16 = 443
var key []byte = client.ToKey(schemaname, host, port)
somedata := []byte{
// Some binary data
}
// Read into schemaname, host and port
client.Key(somedata).Scan(&schemaname, &host, &port)
```

Listing 2: Sample code for key serialization and deserialization

Since keys are implemented as byte arrays, an adapter implementer can manually generate keys in any format they wish. Empirator offers some infrastructure to help with key generation, however: Using the ToKey() and Key.Scan() functions, a number of data types can be transparently converted to and from byte arrays. ToKey() is called with a variable number of arguments, and serializes them to bytes. Currently, ToKey() supports signed and unsigned integers between 8 and 64 bit, strings and booleans. Integers are serialized in big-endian format, while strings are serialized using a 16-byte length field. Key.Scan() allows easy conversion in the opposite direction. Care must be taken to always serialize and deserialize variables in the same order, since no integrity checking is done by the key helper functions. For an example usage of ToKey() and

`Key.Scan()`, see listing 2.

Adapters can also implement toolchaining support. When a certain message is received from the server, an adapter supporting toolchaining should execute a script which generates new data. Examples include downloading new data from online sources, or executing some kind of scan. The update script can receive a number of parameters during execution, which are sent to the binary's standard input. The script to be executed can be passed as the final argument to `NewServerClient()`. If the empty string is passed, toolchaining is not supported.

When implementing an update script, care must be taken to allow updating while an indexing process is running. Indexing of duplicate items due to modification of the data currently being indexed should be avoided at all costs. Also, newly downloaded data should be available to the adapter immediately, without changing any configuration in the adapter. If a scan creates a new database, for example, the adapter must somehow be notified of this newly created database. In the next two chapters, the design and implementation of two currently available adapters will be described in more detail.

4.2.2.1 BGP Adapter

The BGP adapter currently available for the Empirator system is configured to index BGP dumps in the MRT format made available by projects such as Route Views[11]. The Route Views project runs a number of collectors that participate in the BGP protocol, and periodically makes their routing database and BGP interaction publicly available. Every two hours, a new full RIB dump is generated, and every 15 minutes, BGP routing updates received during the last 15 minutes are output.

These output files are stored in the MRT format. MRT is a binary format specified in RFC 6396 [21] for storing BGP protocol dumps. These BGP files are compressed via GZIP to reduce storage requirements. The information contained in these files can be read by using the `bgpdump`[12] tool.

A MRT file can contain a variety of information, depending on the type of entry. The BGP data adapter analyzes three types of entries in a MRT file: Announce, Withdraw and Dump. Announce and Withdraw correspond to the prefix updates carried in a BGP Update message described in chapter 4.2.2.1, while the Dump entry type does not correspond to a BGP message, but describes a special entry for storing the router's current RIB. Thus, Dump entries always describe the current state of the router's routing database, while Update and Withdraw messages correspond to actual received BGP messages.

[11]see `http://www.routeviews.org/`

[12]available at `https://bitbucket.org/ripencc/bgpdump/wiki/Home`

For the purposes of the Empirator system, we map the prefix an announcement refers to to the object, and use the time of the dump as the timestamp, while the rest of the data will be treated as data items. Thus, the BGP adapter supports objects of the `IP` type.

A full description of the BGP adapter data layout can be found in table 4.5.

Column name	Column description
`src_ip`	The source IP of this database entry.
`src_as`	The source AS of this database entry.
`prefix`	The prefix this entry refers to. Used as object of data item.
`as_path`	The AS path this route takes to the target prefix.
`origin`	The origin of this message.
`nexthop`	The next hop address for this prefix.
`localpref`	The local preference value of this entry.
`med`	The multi exit discriminator.
`communities`	The community string for this entry.
`aggregated`	Whether this message was aggregated or not.
`aggregator`	The aggregator of this message.
`type`	Type of entry (Dump, Announce or Withdraw).
`mod_time`	The last modification time of this entry.

Table 4.5: BGP adapter data format

The key used by the BGP adapter for data item identification is the line number and file name where the data is stored, as well as the number of lines used for key batching. The BGP adapter implements key batching by associating a fixed number of entries in the BGP dump file with a single key. In the current implementation of the BGP adapter, the data contained in three adjacent files are associated with a single key.

To allow quicker data retrieval, data is preprocessed upon retrieval: The bgpdump command is called to dump the data in a machine-readable format, the output is split into a certain number of smaller files, and the output is compressed again. This removes the need to open and decompress the large original file. Also, this does not require calling the bgpdump binary during data retrieval. Files are kept in a compressed format because MRT data compresses very well (with a factor of about 10), and keeping uncompressed files would quickly use up a prohibitive amount of storage (a single database dump from a well-connected observer takes about 500 MB of storage space in uncompressed form).

When the BGP adapter receives a `START_INDEX` message from the server (see chapter 4.3 for a complete overview of the protocol), all files with a modification date later than the last indexed date are selected, and index items are generated from the data contained in the MRT file. Each index item contains the prefix contained in the entry as object, and the modification time of the file as timestamp. Keys are generated as described above.

When data is retrieved from the adapter, the received keys are decoded, all keys containing data from the same file are sorted, and data is then read from the file in ascending order. This avoids having to do heavy seeking during key retrieval.

The BGP adapter is set up to do automatic data retrieval using the toolchaining support in the Empirator system. A script is invoked whenever an update is requested by the server. This script downloads the newest BGP dumps from the Route Views project, preprocesses it into smaller files and saves them in a directory accessible to the adapter. When the adapter is indexed, all new data is automatically included, and can be processed by the Empirator system.

4.2.2.2 SSL data adapter

Another adapter was implemented for reading the SSL data collected in [22]. The project scanned a wide variety of hosts, and collected the returned X.509 certificate data. The stored information included both a mapping from IP addresses to X.509 information, as well as a DNS scan, which resulted in a mapping of DNS data to X.509 information. The SSL adapter supports accessing both types of data. SSL scan data is stored in a database, so the primary key of the `hosts` table works well as a key: It identifies a unique data item and allows efficient retrieval. A combination of host and port number is used in keys for IP data, and a combination of hostname and port number for DNS data is used in the DNS data keys. Additionally, the database uses schemas to separate scans from each other. Since a host can be present in multiple scans, the schema will additionally be stored in the key to differentiate between different occurrences of the same object.

Due to the data being stored in a database, indexing is rather simple: Select all hosts from every schema, generate a key and send it to the index. When retrieving, schema name, host and port are extracted from the key, the schema is opened in the database, and the relevant information is selected with a simple SQL statement.

The adapter supports all data that is described in [22].

4.2.3 Frontends

Frontends allows users to access the data contained in the Empirator system. Two types of frontends are currently supported: A web frontend for sighting available data and executing simple queries, and a Python module for exploratory programming and more complex analysis. Both web frontend and the Python module access the Empirator server via a JSON web interface.

URL	Method	Arguments	Description
`/sources`	`GET`	None	Lists names of connected sources.
`/sourceinfo`	`GET`	source=<sourcename>	Returns information about named source.
`/startindex`	`POST`	source=<sourcename>	Starts indexing of the named source.
`/query`	`POST`	query=<query>	Starts a query.
`/indexprogress`	`POST`	identifier=<sourcename>	Returns information about the running indexing process.
`/getdata`	`POST`	identifier=<identifier>	Returns information about the running retrieval process and available data.
`/startupdate`	`POST`	source=<sourcename>, parameters=<parameters>	Starts the toolchaining script on the named source.
`/cancelindex`	`POST`	source=<sourcename>	Cancels the indexing process on the named source.
`/cancelretrieve`	`POST`	identifier=<identifier>	Cancels the retrieve process with the named identifier.
`/filters`	`GET`	None	Lists all available filters.
`/filterdesc`	`GET`	filter=<filtername>	Gets the filter description of the named filter.

Table 4.6: Web HTTP API calls

The JSON interface makes all actions of the Empirator accessible over the network. A complete list of API call with their description and arguments is available in table 4.6. The term "source" is used here instead of adapter, since from the client's view, data comes from some data source that happens to be provided by an adapter.

The `/sources` and `/sourceinfo` calls are used to get an overview of connected sources and available data.

The `/startindex` command initiates indexing a specified source, which can be useful for sources that are not automatically indexed. If an indexing process is started, the state of the process can be retrieved from `/indexprogress`. Available information includes the number of indexed items, any errors that caused termination of the indexing process, whether the process is finished and the start and stop times of the process. An indexing process can be canceled by calling the `/cancelindex` URL.

The `/query`, `/getdata` and `/cancelretrieve` commands allow the same actions for data retrieval operations. However, instead of a source name, they take an "identifier" as argument. The identifier is a server-generated key that is used for processes. Each retrieval process is assigned a separate identifier, and returned to the user when a query is started via `/query`. The `/getdata` command returns meta information about the

Query

```
{"Type": "<querytype>", "Objtype": "<object type>",
  "Limit": 10,
  "FilterNames": ["filter1","filter2"], "FilterArgs": [["args1"],["args2"]],
  "Query": "<selector>"}
```

Selector

```
{"QueryType": "<selector type>",
  "QueryObject": "<query object>",
  "Left": "<left subselector>", "Right": "<right subselector>"}
```

Table 4.7: Query JSON format

retrieval process (such as number of retrieved items, start and stop time, and any errors encountered during retrieval) and the data that has been collected so far. After the client has retrieved the data, it is deleted on the server side to conserve memory. The `/getdata` command thus has a side effect, unlike the `/indexprogress` command.

The `/startupdate` command starts the update script for the given adapter, passing the parameters (a string list) to the script's standard input.

Finally, the `/filters` and `/filterlist` list the available filters and allow retrieving filter information for the specified filter.

Queries and data are also encoded in a JSON format. Queries use the JSON format described in table 4.7. The "<query type>" parameter is one of the supported query types (Limited, Standard, Count or Filtered). The "Limit" and "FilterNames" and "FilterArgs" parameters need only be present if the query is of the corresponding type. If the "Query" elemnt is omitted as well, the query applies to all the items in the database. The "<selector type>" argument is one of the selector names described in table 4.3. The "Left" and "Right" arguments are required if the selector is a boolean selector, such as OR, while the "QueryObject" argument is required for all other selectors.

The data format used by the web frontend is similar in structure to the data storage used internally by the system, described in table 4.4. Data is returned as a JSON object, which has the objects contained in the data as attributes. These attributes contain another JSON object, which have the adapter names which returned data for this object as attributes. Finally, the value of each of these attributes is a list of JSON objects having a "Time" and "Data" object, where the "Time" property contains the time of observation of the data, and the "Data" object contains a list of strings corresponding to the data returned for this object. Additionally, the list of column names for each adapter is contained in the "Column" attribute of the encoded data, which allows clients to access

```
{"Columns": {
    "BGP_Adapter": ["prefix","origin","src_as","..."],
    "SSL_Adapter": ["akid", "ca", "cert","..."]},
 "Data": {
     "8.8.8.8/32": {
         "SSL_Data": [
             {"Time":"2015-05-03T17:22:33Z",
              "Data": ["1","CaCert.org","--BEGIN CERTIFICATE...","..."]},
             {"Time":"...",
              "Data": ["..."]}]},
     "8.8.8.0/24": {
         "BGP_Data": [
             {"Time":"2015-06-30T17:30:22Z",
              "Data": ["8.8.8.0/24", "IGP", "35","..."]}]}}}
```

Listing 3: Data JSON format

the columns in the returned data by name. An example of this structure can be seen in listing 3.

4.2.3.1 Web Frontend

While the JSON interface of the index can be used manually via tools such as `curl` or `wget`, this is not very comfortable for deeper exploration of the data offered by the system.

In order to provide a solution for low-effort exploration of available data sources and data, a web interface was implemented in JavaScript, using the REST interface described in the previous section.

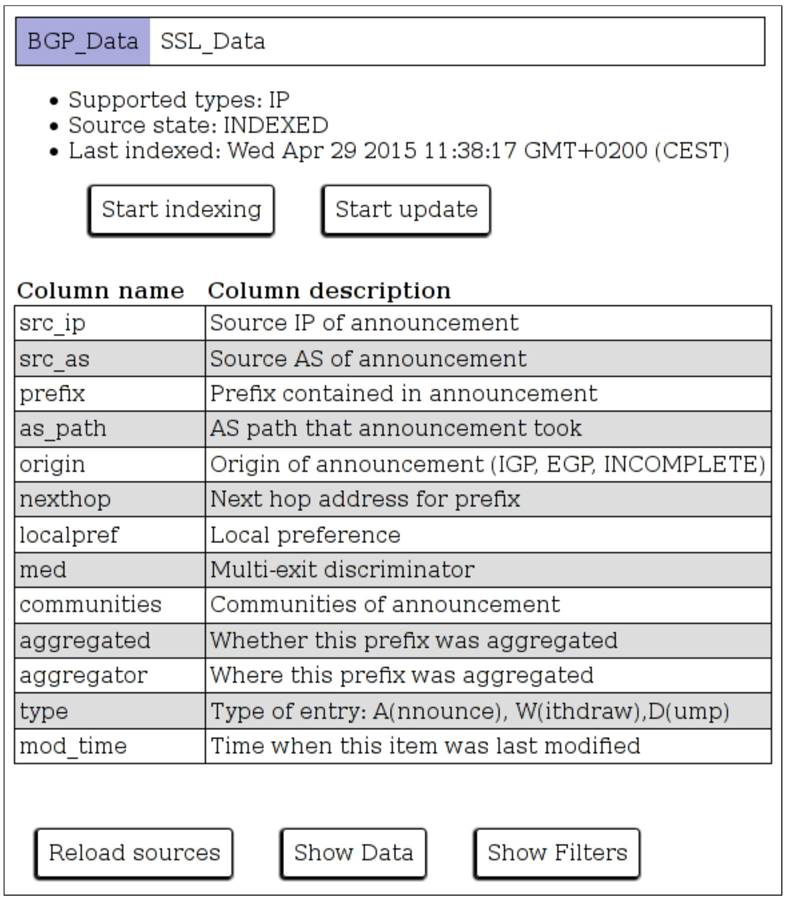
The web interface allows a user to look at the structure of data contained in the different data sources, and also retrieve samples of the data.

In figure 4.4, a screen shot of the data sighting interface is shown. In the top navigation bar, all connected sources are listed. A click on the source name displays status information about the sources in the main display area.

The information displayed contains:

- The supported object types (a subset of IP,DNS and AS)
- The current source state (see chapter 4.2.1)
- When this source was last indexed.

If the source is currently being indexed, statistics about the current index progress are displayed. Indexing and toolchaining of the source can be started with the “Start



- Supported types: IP
- Source state: INDEXED
- Last indexed: Wed Apr 29 2015 11:38:17 GMT+0200 (CEST)

Column name	Column description
src_ip	Source IP of announcement
src_as	Source AS of announcement
prefix	Prefix contained in announcement
as_path	AS path that announcement took
origin	Origin of announcement (IGP, EGP, INCOMPLETE)
nexthop	Next hop address for prefix
localpref	Local preference
med	Multi-exit discriminator
communities	Communities of announcement
aggregated	Whether this prefix was aggregated
aggregator	Where this prefix was aggregated
type	Type of entry: A(nnounce), W(ithdraw),D(ump)
mod_time	Time when this item was last modified

Figure 4.4: The data sighting screen

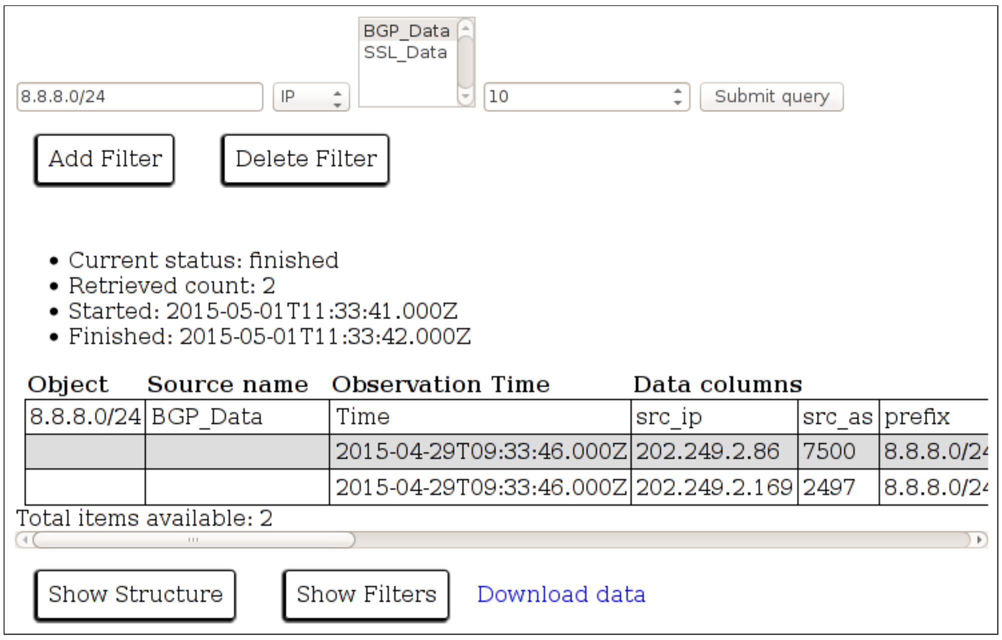
indexing" and "Start update" buttons.

Below the source information, the data structure of the source is displayed. Each element of a source's data is listed with its short name and the description given to it by the adapter implementer.

On the bottom row, buttons for updating the display, showing the data selection interface and showing the filter listing interface are shown. The filter listing interface shows a list of available filters with the description configured by the filter author.

Clicking the "Show Data" button takes the user to the data selection screen shown in figure 4.5.

In the data selection screen, sources can be searched for data. The input fields at the top of the interface are provided for object selection: The object is entered in textual form in the first text field, the type of item needs to be selected in the selection box, and one or more sources to retrieve data from can be selected in the selection box. A maximum number of items to retrieve can be selected in the number entry box.



8.8.8.0/24 IP 10

- Current status: finished
- Retrieved count: 2
- Started: 2015-05-01T11:33:41.000Z
- Finished: 2015-05-01T11:33:42.000Z

Object	Source name	Observation Time	Data columns		
8.8.8.0/24	BGP_Data	Time	src_ip	src_as	prefix
		2015-04-29T09:33:46.000Z	202.249.2.86	7500	8.8.8.0/2
		2015-04-29T09:33:46.000Z	202.249.2.169	2497	8.8.8.0/2

Total items available: 2

Figure 4.5: The data selection screen

Additionally, the “Add Filter” and “Delete Filter” buttons allow the application of filters to the search result. Multiple filters and arguments can be chosen for processing the resulting data.

On clicking the “Submit query” button, the data for the specified object is retrieved and displayed to the client.

Below the filter buttons, the state of the retrieval process is displayed. It is updated periodically while the retrieval is in progress.

Below that is the data display. Data is displayed sorted by object and source name. For each source, the available data for the object is displayed in the table columns. Data is sortable by observation time or data values by clicking the “Time” column or one of the data columns. Below the data table, the total number of items available in the database is displayed. Finally, the buttons in the bottom row allow switching the user interface back to the data sighting screen, or to a filters display. The “Download data” link allows the user to download the data of the last query as a JSON file.

4.2.3.2 Python Module

For more complex analysis of the available data, a Python module that allows easy access to the data available in the Empirator system is provided. The module allows convenient access to all commands implemented in the JSON interface. Reasons for the choice of Python include its wide usage in the scientific programming community, and very good support for graphing and multi-dimensional arrays.

The Python module offers object-oriented access to retrieve and indexing progress, retrieved data encoded into python structures with simpler access methods (such as asking for data by column name, or retrieving all data for a certain object) and function-based construction of queries. Both synchronous and asynchronous data access are available for queries. If data is retrieved synchronously, a complete set of data is constructed by the module, and returned to the client after the query has finished. This allows the client to process a full set of data. Asynchronous access can be used to receive data in a "piecemeal" fashion. Whenever data is available, a callback function is called with the retrieved data. This is repeated until the query has finished.

The Python module offers both a function-oriented interface as used in the example code in the "comfort" module, and an object-oriented interface in the "access" module. The object-oriented module offers a thin abstraction layer over the JSON interface: An access object is constructed, and operations Empirator system can be executed by calling methods on that access object. The "comfort" module maintains a single global connection to the system, and functions allow interaction with the Empirator system. Additionally, the "comfort" module provides a number of convenience methods for retrieving data both synchronously and asynchronously, as described above.

A sample program using the comfort interface can be seen in listing 4.

4.3 Communication protocol

To enable communication between data sources and the central server, a simple communication protocol was devised. Because the networking layer of the system is based on the ZeroMQ messaging library, implementation effort was low. The data source or adapter will be referred to as client in the following section, because the adapter-index connection implements a client-server architecture. This "client" should not be confused with the use of "client" in the previous chapter, which describes a user of the Empirator system.

Using the DEALER/ROUTER[13] architecture of ZeroMQ allows implementation of a regular server/client infrastructure using an abstracted address called "identity". The DEALER/ROUTER configuration makes it possible for server and client to communicate asynchronously. Data framing, non-blocking IO, buffering and routing concerns are taken care of as well by the ZeroMQ implementation, which means that actual implementation of the protocol mostly consisted of devising a serialization scheme for messages from and to byte arrays.

[13] `http://zguide.zeromq.org/page:all#Shared-Queue-DEALER-and-ROUTER-sockets`

```
#!/usr/bin/python

from comfort import *
import query as q
import objects as obj
import re
from collections import Counter

connect("localhost:5556")
query = q.Query(obj.OBJECT_IP, q.SourceExact("BGP_Data"))
as_count = Counter()

def count_it(retrieved):
    data = retrieved.data()
    for ob in data.objects():
        for source, objdata in data.object_data(ob):
            for (timestamp, final_items) in objdata:
                aspath = final_items[3]
                for asnum in re.split("\s+|,",aspath):
                    asnum = asnum.strip("[{}]|")
                    try:
                        as_count[int(asnum)] += 1
                    except ValueError:
                        print "Invalid AS number %s" % asnum

call_with_data(query, count_it)

print as_count.most_common(10)
```

Listing 4: Sample program using the Python module

Messages are prefixed with their type, and the rest of the message is deserialized according to the message type. Note that "administrative" data such as serial numbers or a total length field are not included in the message — ZeroMQ takes care of message framing, and serial numbers are already used in the TCP protocol used by ZeroMQ.

Ten types of messages are used in the Empirator system, listed in table 4.8.

The messages `PING` and `ACK` are used for client and server liveness checks: The server sends a ping to each connected adapter every two minutes. If an ACK message is not received from the adapter in a certain time (30 seconds by default), the adapter is seen as disconnected. If the adapter has not received a ping from the server in a certain time interval (5 minutes by default), the server is presumed dead, and the client exits. These intervals were chosen to not generate an undue amount of network traffic, but still allow checking of connectivity. A standard disconnection mechanism is not implemented for

Message type	Description
REGISTER_SOURCE	Registers a new source with the index.
START_INDEX	Requests a data source to send index information.
REQUEST_DATA	Requests a data source to send data for the specified keys.
PING	Request an ACK from the client as soon as possible.
ACK	Reply for PING requests.
RESULT	Used for indicating completion of indexing or retrieval, or cancelling indexing or retrieval.
INDEX_ITEM	Message for sending a single index item to the index.
INDEX_REQUEST	Used by adapters to request being indexed.
ITEM_DATA	Message for sending a single data item to the index.
GET_INDEXITEMS	Used by the server to request a number of index items from the client.
START_UPDATE	Start the toolchaining script on the client.

Table 4.8: Message Types

clients, as clients should only disconnect from the server by either user interference or network problems.

The core messages of the protocol are `REGISTER_SOURCE`, `START_INDEX` and `REQUEST_DATA`. `REGISTER_SOURCE` allows sources to register with the server. A `REGISTER_SOURCE` message contains a list of the object types supported by the adapter, a short description of the data as well as the source name. This allows new adapters to connect to the server, and immediately participate in the system, without any configuration on the server side.

`START_INDEX` alerts an adapter that it should start indexing. The message contains a timestamp which indicates when this adapter was last indexed. No index items about observations before that time point should be generated, but all data created since then should be sent to the server.

The `GetIndexItems` function of the adapter is called, and index items are created from the adapter's data. A `GET_INDEXITEMS` message is then sent by the server to request a certain number of index items. When the client receives this message, it sends the requested number of index items (or a `RESULT` message indicating that the indexing is finished) to the server. This "pull" design was chosen to prevent overload when many clients are indexing at once.

When a `REQUEST_DATA` message is received by the client, the keys are extracted from the message. The keys are passed to the `GetIndexItems` function, which retrieves the associated data from its storage. That data is then sent to the server as a series of `INDEX_ITEM`s.

Finally, when a `START_UPDATE` message is received by the client, the toolchaining script is started, if one is configured. If the script runs successfully, the client then indicates arrival of new data using the `INDEX_REQUEST` data. The server receives this message and starts an indexing process on the adapter when convenient.

An overview of the messages exchanged during the different phases of the system can be seen in figure 4.6.

At any time during messaging exchanges like indexing or data retrieval, a failure message can be sent, either by the client or the server. If the server receives a failure message from the client, it aborts the current indexing or retrieval operation, and indicates an error to the system user. If the server sends a failure message to the client, the client aborts its current process (be it retrieval or indexing).

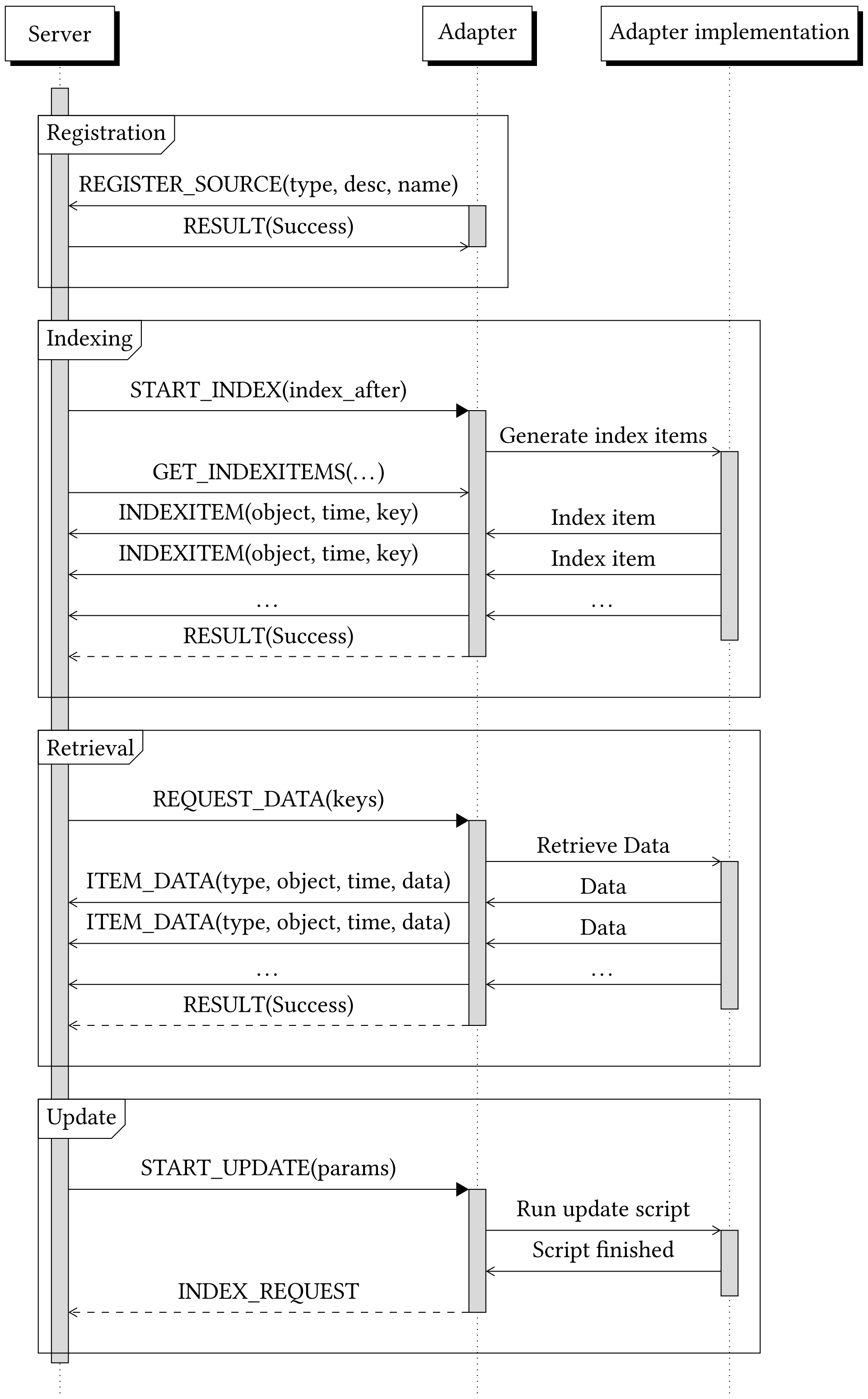


Figure 4.6: Messages exchanged during registration, indexing and data retrieval.

Chapter 5

Evaluation

5.1 Performance Evaluation

Performance is an important factor of the Empirator system. In this chapter, we will show some metrics describing the performance characteristics of the Empirator system.

5.1.1 Index Database Performance

The index database is the central component governing the speed of the system. Therefore, we executed some queries for evaluating the performance of the database.

At the time these analysis were executed, the database contained a total of 520 million items, with information for 73.5 million distinct IP networks and 3.1 million distinct DNS names recorded.

This information was generated from the data described in [22], which contains 7 full IPv4 SSL scans and 8 Alexa Top 1 Million SSL scans. The BGP data in the system was generated by reading MRT data dumps from the routeviews project collected by the WIDE collector[1]. The database dumps and routing updates of one month were analyzed. After importing initial BGP data, a continuous update was configured using the toolchaining functionality described in chapter 4.2.2.1, and up-to-date BGP data was being important constantly.

Figure 5.1 shows a graph indicating the indexed items during a long-running indexing progress. Indexing data for both the BGP adapter and the SSL adapter is shown. Both indexing processes were running concurrently.

The X axis indicates the total amount of items in the database, while the Y axis how many items were added at that time point. Thus, the whole graph can be seen as an

[1]See `http://archive.routeviews.org/route-views.wide/bgpdata/`

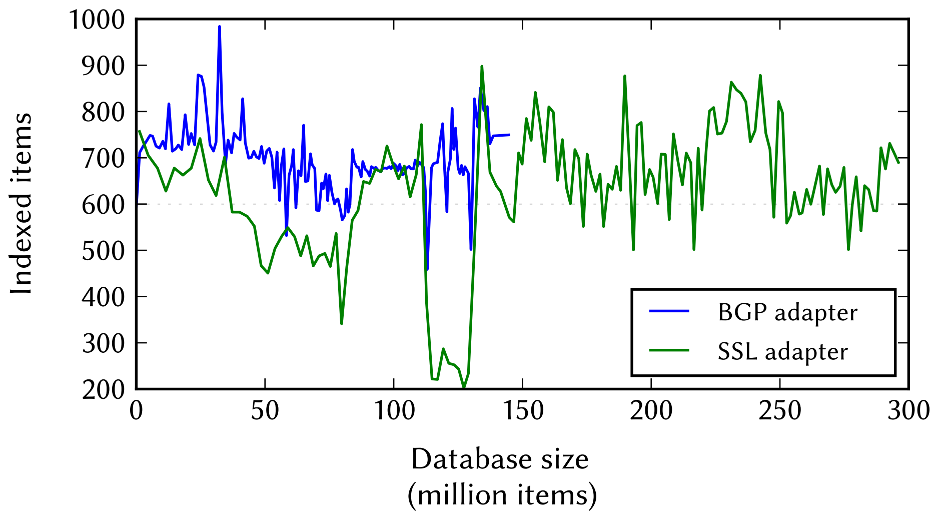


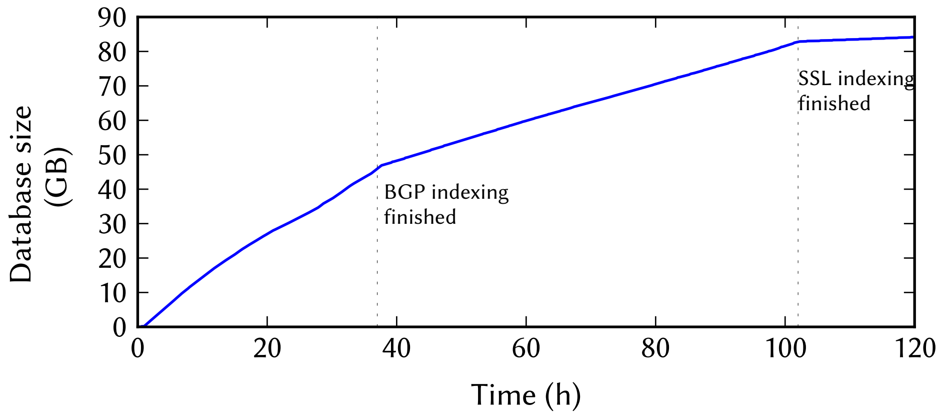
Figure 5.1: Indexing speed vs. items in database

indicator of indexing speed. Indexing speed decreases while the first 50 million items are indexed, which stems from the insertion of new objects into the database (refer to chapter 4.2.1.3 for the database scheme). The speed decrease at 125 million items for the SSL adapter was likely caused by the database being used by other researchers. The SSL adapter is influenced by database usage more strongly, since the SSL data is stored in the same database as the index database.

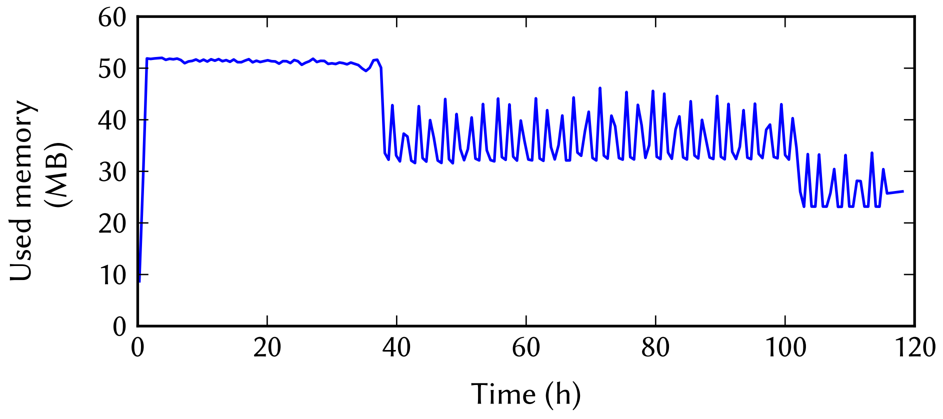
In general, it can be seen that adding additional items to the index database does not decrease indexing speed. No downward trend in the number of added items can be discerned in the graph, and indexing speed remains at a minimum of 600 items/s at most times.

In figure 5.2a, the size of the index database is shown over time. The run times of the indexing processes can be seen in the diagram: The database size grows more quickly while both the BGP and SSL indexing processes are running. The database grows less quickly after initial BGP indexing is finished. After SSL indexing is finished, the ongoing BGP indexing increases the database size slowly. Total database size is about 80 GB after indexing has finished. The combined size of the datasets (SSL database and uncompressed BGP dumps) is about 220 gigabyte.

Finally, the memory usage of the Empirator index over the time of indexing is shown in figure 5.2b. The system quickly reaches a plateau of about 50 megabyte memory usage. This decreases to about 38 to 40 megabyte after BGP indexing is finished, and decreases further to about 35 megabyte after SSL indexing is finished. The pipelined structure of the Empirator system allows this low memory usage: Items are dispatched



(a) Database size over time



(b) Memory used by the index core over time

Figure 5.2: Database and memory stats during indexing

from adapter to index to database in a continuous fashion, so only a small amount of memory is needed.

5.1.2 Protocol Performance

Performance of the network protocol is not a prime concern for control messages (such as `START_UPDATE`, `REGISTER_SOURCE` or `PING`), but during indexing, the performance of the protocol becomes much more important. The indexing protocol causes the adapter to send a large number (on the order of millions) of very small messages (about 30 byte for an index message containing a single key, object and timestamp). However, ZeroMQ batches messages, reducing the impact of such a large number of small packets. A

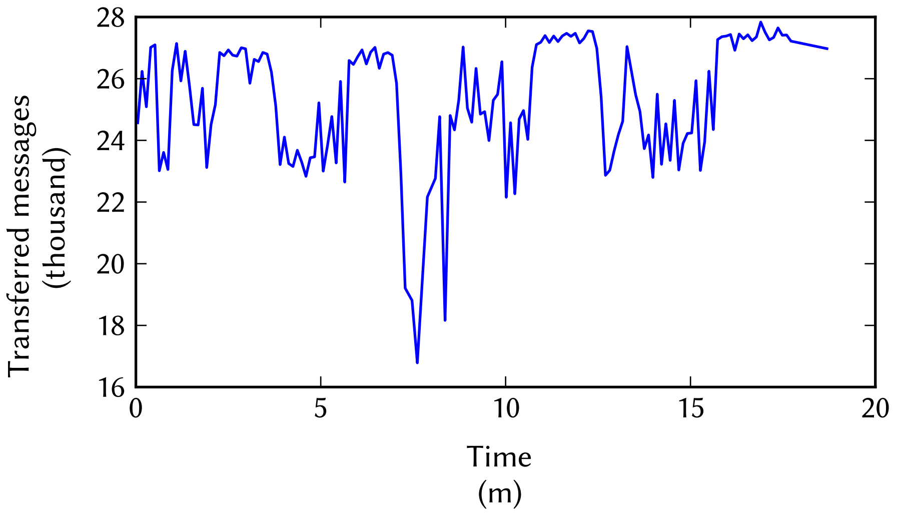


Figure 5.3: Protocol speed

Type	Total	Average	Median	Standard deviation
IP address, single	637 s	6.37 s	5 s	3.88 s
IP networks, single	6403 s	64.03 s	64.5 s	46.5 s
DNS names, single	37 s	0.37	0.01 s	0.45 s

Table 5.1: Retrieval timings per request. Total number of items was 100

diagram of protocol throughput in a synthetic test case sending a large number of index items can be seen in figure 5.3. The average number of messages transferred per second is about 25000 messages, with the drop at 400 seconds caused by non-benchmark use of the system. Average throughput during the run was about 1.75 MByte/s.

5.1.2.1 Retrieval Performance

In order to evaluate retrieval performance of the Empirator system, several analyses were executed. First, a random subset of 100 IPs was drawn from the Empirator database, and data for those IPs was retrieved from the system using the Python module. The whole test took about 10 minutes, which means an average time of 6 seconds for a single request.

These tests were repeated with random IP networks and random DNS names. From the gathered data, it is immediately apparent that retrieval time differs greatly, depending on the adapter used. This is caused by the storage of IP data in a database, which allows quick random access to the data, while BGP data has to be extracted from a number of compressed files. The quick retrieval time of DNS data stems from the smaller data volume contained in the system as well as DNS data being stored in a database as well.

5.2 Data Analysis

To evaluate the suitability of the Empirator system for combined data analysis, an experiment combining SSL and BGP data was executed. The experiment measured the correlation between SSL certificates and their expiry dates, and the availability of routing data for those certificates.

The experiment was set up as follows: A list of 1000 randomly chosen IP addresses was selected from the system (we chose a limit of 1000 addresses owing to time constraints). Then, for each IP, the system was searched for BGP announcements for prefixes containing this IP using a COUNT query. If no announcement was found for these IP addresses, the address was not reachable. For all IPs in the system that were not reachable, certificate data was selected from the system. Finally, the certificate data, expiry date and issuer were analyzed.

```
count_bgp_entries = query.Count(object.OBJECT_IP,
        query.And(
            query.And(
                query.And(
                    query.TimeAfter(datetime(2015,5,11,17,59)),
                    query.TimeBefore(datetime(2015,5,11,18,1))),
                query.SourceExact("BGP_Data")),
            query.IPIsContained(ip)))

retrieve_ssl_data = query.Query(objects.OBJECT_IP,
                            query.And(
                                query.SourceExact("SSL_Data"),
                                query.IPExact(ip)))
```

Listing 5: Example queries used during the experiment

The queries used during the experiment can be seen in listing 5. The first query checks whether the IP specified as argument is reachable. This is done by selecting a timestamp where a BGP database dump dump took place (found via the web interface by searching BGP data), and limiting the data to data retrieved from the BGP adapter. If the number of items in the database is zero, no routing entry for that IP was contained in the BGP database dump, which means the IP address was not reachable at that time. After that, the certificate data for each unreachable IP was retrieved with the second query. By restricting the time, we retrieve only entries from the latest SSL scan. Finally, we check the expiry date of the retrieved certificate data, and output any certificates that are still valid.

The results we found were not surprising: Most certificates had expiry times well before the time of our BGP dump, thus not allowing any conclusions about the reachability. This could be rectified by importing historical BGP data into the system, and querying for BGP data around the time of expiry of these certificates.

A single certificate was found having an expiry date in 2040, while not being reachable any more. The subject of the certificate is “CN=r4-u14.micfo.com”, with micfo.com being a service provider. Possibly this certificate stemmed from a misconfigured external server that was not meant to be exposed to the public internet.

It will be interesting to extend the results of this study with a more thorough experiment using the Empirator system.

Chapter 6

Conclusion

6.1 Future Work

While the Empirator system as described in this work fulfills the requirements outlined very well, several avenues are available for extending use of the Empirator system.

First, and chiefly, the implementation of additional adapters. The combined amount of data available in the Empirator system is its primary asset. Increasing the number of adapters thus greatly increases the utility of the Empirator system. Using the support infrastructure outlined in chapter 4.2.2, implementing an adapter is relatively simple, and should be possible for a wide array of data storage methods.

In regard to features, the Empirator system provides all features necessary for the described use cases. However, several improvements would allow wider deployment of the Empirator system.

Cryptography support for both adapters and users of the system would allow deployment of the system over untrusted network. The ZeroMQ library supports asymmetric cryptography for both authentication and encryption, which would allow adapters and servers to authenticate each other. Securing the web interface would be accomplished by deploying an SSL-enabled proxy to secure the JSON web frontend, and configuring the server serving the web interface to employ HTTPS as well.

By implementing a role- or user-based authentication system and rights management system, sensitive data could be stored in the Empirator database, without the risk of having unauthorized users access that data. This would require extension of the index database with user and role information.

In general, the small code base and flexible implementation of the Empirator system allows easy improvements for a range of features of the system, such as extending the available query formats, optimizing adapter performance, providing more functionality in the Python access module or extending the web interface (graphically and functionally).

Of course, using the functionality of the Empirator system for combined analysis of data opens wide possibilities for further research. The analyses implemented in chapter 5.2 only exercised a small part of the experiments possible with the Empirator system.

6.2 Summary

The goal of this work was the design of a system that allowed centralized organization and retrieval of network measurement data. Additional goals for the system were simplicity and ease of extension, convenience for the researcher and adequate performance for interactive use.

To reach these goals, we made a number of important decisions about the design of the system. Several key properties were identified during the design process: Decentralized data storage allows data to remain in the original system, and access via a custom access component allows access to wide variety of data sources. A flexible data format enables storage and retrieval of arbitrary data limiting scope of the system. Limited data selection options allow the system to perform well during data retrieval, and still allow most interesting operations. Filtering allows arbitrary processing and extension of the data during retrieval, and toolchaining enables simple adaption of experiments for continuous data collection.

The system was then implemented, keeping the aforementioned design decisions in mind. The implementation consisted of a central component, containing functionality to index data, retrieve data via an extensive set of queries, automate collection of data and augment retrieved data via configurable scripts. Additionally, adapters for two networking experiments were implemented: An adapter for accessing SSL scan data contained in a database, and a second adapter for accessing data contained in BGP dump data. For communication between the adapters and the central component, a simple network protocol was designed and implemented. Finally, for accessing the functionality of the system, a web user interface and a python module for programmatic access were implemented.

That system was then evaluated, proving satisfactory performance for both retrieval and indexing. Finally, a simple example experiment correlating host reachability with certificate expiry dates was was implemented, finding several certificates on hosts no longer reachable, but valid nonetheless.

In conclusion, the final implementation of the system fulfills the goals we set out to achieve in the beginning of this work. The Empirator system allows running experiments using existing data with little effort, while being easy to extend and understand.

The author believes that the Empirator system will be a valuable tool in a network researcher's toolbox.

Bibliography

[1] Z. Durumeric, E. Wustrow, and J. A. Halderman, "Zmap: Fast internet-wide scanning and its security applications." in *USENIX Security*. Citeseer, 2013, pp. 605–620.

[2] O. Gasser, R. Holz, and G. Carle, "A deeper understanding of SSH: results from internet-wide scans," in *Network Operations and Management Symposium (NOMS), 2014 IEEE*. IEEE, 2014, pp. 1–9.

[3] R. Holz, L. Braun, N. Kammenhuber, and G. Carle, "The SSL landscape: a thorough analysis of the x. 509 pki using active and passive measurements," in *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. ACM, 2011, pp. 427–444.

[4] Y. Rekhter, T. Li, and S. Hares, "A Border Gateway Protocol 4 (BGP-4)," RFC 4271 (Draft Standard), Internet Engineering Task Force, Jan. 2006, updated by RFCs 6286, 6608, 6793. [Online]. Available: http://www.ietf.org/rfc/rfc4271.txt

[5] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," RFC 5280 (Proposed Standard), Internet Engineering Task Force, May 2008, updated by RFC 6818. [Online]. Available: http://www.ietf.org/rfc/rfc5280.txt

[6] M. Allman, E. Blanton, and W. Eddy, "A scalable system for sharing internet measurements," in *Proc. PAM*. Citeseer, 2002.

[7] C. Shannon, D. Moore, K. Keys, M. Fomenkov, B. Huffaker *et al.*, "The internet measurement data catalog," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 5, pp. 97–100, 2005.

[8] P. A. A. Gutièrrez, A. Bulanza, M. Dabrowski, B. Kaskina, J. Quittek, C. Schmoll, F. Strohmeier, A. Vidacs, and K. S. Zsolt, "An advanced measurement meta-repository," in *Proceedings of 3rd International Workshop on Internet Performance, Simulation, Monitoring and Measurement*, 2005.

[9] M. Dabrowski, J. Sliwinski, and F. Strohmeier, "The MOME workstation as a platform for automatic analysis of measurement data," in *Distributed Cooperative Lab-*

oratories: Networking, Instrumentation, and Measurements. Springer, 2006, pp. 409–424.

[10] B. Trammell, P. Casas, D. Rossi, A. Bär, Z. Ben-Houidi, I. Leontiadis, T. Szemethy, and M. Mellia, "mPlane: an intelligent measurement plane for the internet," *IEEE Communications Magazine, Special Issue on Monitoring and Troubleshooting Multi-domain Networks using Measurement Federations*, vol. 42, 05/2014 2014.

[11] A. Hanemann, J. W. Boote, E. L. Boyd, J. Durand, L. Kudarimoti, R. Łapacz, D. M. Swany, S. Trocha, and J. Zurawski, "perfSONAR: A service oriented architecture for multi-domain network monitoring," in *Service-Oriented Computing-ICSOC 2005.* Springer, 2005, pp. 241–254.

[12] E. Boschi, S. D'Antonio, P. Malone, and C. Schmoll, "Intermon: An architecture for inter-domain monitoring, modelling and simulation," in *NETWORKING 2005. Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; Mobile and Wireless Communications Systems.* Springer, 2005, pp. 1397–1400.

[13] D. Leonard and D. Loguinov, "Demystifying service discovery: implementing an internet-wide scanner," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement.* ACM, 2010, pp. 109–122.

[14] J. Heidemann, Y. Pradkin, R. Govindan, C. Papadopoulos, G. Bartlett, and J. Bannister, "Census and survey of the visible internet," in *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement.* ACM, 2008, pp. 169–182.

[15] "Internet Census 2012 — Port Scanning /0 Using Insecure Embedded Devices," Mar. 2013, available at http://census2012.sourceforge.net/paper.html.

[16] Z. Team and Rapid7, "Internet-wide scan data repository," online, 03 2015, http://scans.io.

[17] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining your Ps and Qs: Detection of widespread weak keys in network devices." in *USENIX Security Symposium*, 2012, pp. 205–220.

[18] N. Provos and P. Honeyman, "ScanSSH: Scanning the internet for SSH servers." in *LISA*, 2001, pp. 25–30.

[19] P. Eckersley and J. Burns, "An observatory for the SSLiverse," *URL: www.eff.org/files/DefconSSLiverse.pdf*, 2010, available at http://www.eff.org/files/DefconSSLiverse.pdf.

[20] R. Pike, "Go at Google: Language Design in the Service of Software Engineering," online, 10 2012, http://talks.golang.org/2012/splash.article.

[21] L. Blunk, M. Karir, and C. Labovitz, "Multi-Threaded Routing Toolkit (MRT) Routing Information Export Format," RFC 6396 (Proposed Standard), Internet Engineering Task Force, Oct. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6396.txt

[22] R. Holz and G. Carle, "Reproducing and reusing: Documenting methodologies, uses and limitations by example of a X.509 data set," unpublished.