BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

EFFECTIVE LARGE-SCALE COLLECTION OF INFORMATION RELATED TO DOMAIN NAMES

EFEKTIVNÍ ROZSÁHLÝ SBĚR INFORMACÍ O DOMÉNOVÝCH JMÉNECH

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR AUTOR PRÁCE	Bc. ONDŘEJ ONDRYÁŠ
SUPERVISOR VEDOUCÍ PRÁCE	Ing. RADEK HRANICKÝ, Ph.D.

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

154543

Institut:	Department of Information Systems (DIFS)
Student:	**Ondryáš Ondřej, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Computer Networks
Title:	**Efficient Large-scale Collection of Information Related to Domain Names**
Category:	Networking
Academic year:	2023/24

Assignment:

1. Research available sources of domain name-related information (e.g., DNS, WHOIS, RDAP, TLS certificates, geolocation information, reputation systems, etc.) that could be used for malicious domain classification purposes.
2. Get acquainted with the current state of research on malicious domain detection under the FETA project.
3. Get acquainted with techniques for parallel and distributed big data processing.
4. In consultation with the supervisor, design a solution for efficient collection and storage of information on large numbers of domain names. The collected data needs to be transformed into a form suitable for machine learning.
5. Implement the proposed solution and optimize it for performance and efficient use of available computing capacity.
6. On a sufficiently large set of domains, experimentally verify the applicability of the designed solution and evaluate the obtained results.

Literature:

- Han, Jiawei, Jian Pei, and Hanghang Tong. "Data Mining: Concepts and Techniques." Morgan Kaufmann, 2022.
- Hajaj, Chen, Nitay Hason, and Amit Dvir. 2022. "Less Is More: Robust and Novel Features for Malicious Domain Detection" *Electronics* 11, n. 6: 969.
- Torroledo, Ivan, Luis David Camacho, and Alejandro Correa Bahnsen. "Hunting malicious TLS certificates with deep neural networks." In *Proceedings of the 11th ACM workshop on Artificial Intelligence and Security*, pp. 64-73. 2018.
- Shi, Yong, Gong Chen, and Juntao Li. "Malicious domain name detection based on extreme machine learning." *Neural Processing Letters* 48.3, pp. 1347-1357. 2018.

Requirements for the semestral defence:
Points 1 to 4.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Hranický Radek, Ing., Ph.D.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	31.7.2024
Approval date:	30.10.2023

Abstract

This thesis presents a software solution that provides fast data collection and feature extraction for the purpose of detecting malicious domain names using machine learning. It introduces the FETA DomainRadar research project targeted at developing a system for assessing domain name maliciousness. It discusses various sources of information that proved helpful for the task. It elaborates the system's design and presents its crucial component for collecting and processing data that can be used to evaluate domain names captured in monitored high-traffic networks in real time and to build large training datasets effectively. Based on Apache Kafka, the system is designed to allow horizontal scalability in distributed deployments, with experiments showing massive improvements in throughput when multiple instances cooperate. The system collected data from eight external sources for 400,000 domain names in about 4 hours, reaching the average throughput of 28 domain names per second. It was deployed in the CESNET academic network, where it steadily collected and processed data at 9.56 domain names per second.

Abstrakt

Tato práce představuje programové řešení poskytující rychlý sběr dat a extrakci příznaků pro účely detekce škodlivých doménových jmen s využitím strojového učení. Představuje výzkumný projekt FETA DomainRadar, jehož cílem je vývoj systému pro vyhodnocování škodlivosti doménových jmen. Pojednává o různých zdrojích informací, které se v této úloze osvědčily. Upřesňuje návrh tohoto systému a prezentuje jeho klíčovou část pro sběr a zpracování dat, kterou lze použít pro pro vyhodnocování doménových jmen zachycených v reálném čase v sítích s velkým provozem, ale také pro efektivní sestavování rozsáhlých trénovacích datových sad. Systém na bázi platformy Apache Kafka je navržen tak, aby umožňoval nasazení v distribuovaném prostředí, a byl tak horizontálně škálovatelný. Provedené experimenty ukazují významný nárůst propustnosti systému při kooperaci několika instancí. Systém zvládl nasbírat data z osmi externích zdrojů pro 400 000 doménových jmen přibližně za 4 hodiny, čímž dosáhl průměrné propustnosti 28 doménových jmen za sekundu. Poté byl nasazen v akademické síti CESNET, kde bez obtíží sbíral a zpracovával data pro 9,56 doménových jmen za sekundu.

Keywords

domain name, DNS, WHOIS, RDAP, IP, TLS, certificates, reputation systems, NERD, big data, data collection, Apache Kafka, Kafka Streams, distributed computation, phishing, malware, classification, detection, feature extraction

Klíčová slova

doménové jméno, DNS, WHOIS, RDAP, IP, TLS, certifikáty, reputační systémy, NERD, velká data, sběr dat, Apache Kafka, Kafka Streams, distribuované výpočty, phishing, škodlivý obsah, klasifikace, detekce, extrakce příznaků

Reference

ONDRYÁŠ, Ondřej. *Effective Large-scale Collection of Information Related to Domain Names.* Brno, 2024. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Radek Hranický, Ph.D.

Rozšířený abstrakt

Internet, jakožto neodmyslitelná součást života dnešní společnosti, je přirozeně úrodnou půdou pro různé formy škodlivých aktivit mířených na jeho uživatele. Útočníci využívají techniky jako phishing a malware k získání přístupu k citlivým informacím, heslům a účtům. V případě phishingu oběti samy poskytnou své přihlašovací údaje falešným stránkám, které věrohodně napodobují legitimní služby. Malware je škodlivý software, který se může nainstalovat na zařízení oběti a umožňuje útočníkům například krást data nebo napadené zařízení ovládat. V boji proti těmto hrozbám je kromě vzdělávání uživatelů vhodné využívat technické prostředky, které dokáží takové útoky detekovat a případně jim zabránit.

Tato práce byla řešena v rámci výzkumného projektu „Analýza šifrovaného provozu pomocí síťových toků“ (Flow-Based Encrypted Traffic Analysis, FETA) na Fakultě informačních technologií VUT v Brně. Skupina DomainRadar, která je součástí projektu, vyvíjí systém, který bude efektivně detekovat potenicální hrozby prostřednictvím klasifikace „škodlivosti“ doménových jmen. Klasifikaci bude provádět na základě různorodých dat o jednotlivých doménových jménech, která získá z externích zdrojů.

Práce představuje komplexní softwarové řešení stěžejní části systému DomainRadar, která umožňuje rychlý sběr dat o doménových jménech. Ze získaných dat poté extrahuje příznaky, které systém využije pro klasifikaci. Kromě těchto komponent práce významně přispívá k celkovému návrhu architektury systému DomainRadar. Rozpracovává konceptuální návrh vytvořený v týmu do podoby, ve které jsou jasně definované datové toky mezi jednotlivými částmi systému a použité technologie. Navržený subsystém je schopen zpracovávat velké objemy vstupních dat v reálném čase a umožňuje horizontální škálovatelnost v distribuovaném prostředí. Je postaven na platformě Apache Kafka, která poskytuje škálovatelnost a odolnost vůči chybám.

Zdroje dat pro klasifikaci škodlivosti domén

První část práce se zabývá analýzou různých zdrojů dat, které mohou poskytnout užitečné informace pro klasifikaci škodlivosti doménových jmen. Druhá kapitola naznačuje, jak dosavadní výzkum přistupoval k úloze klasifikace na základě dat z externích zdrojů. Shrnuje různé charakteristiky, neboli příznaky doménových jmen, které se v literatuře osvědčily. Čtvrtá kapitola podrobně zkoumá několik konkrétních zdrojů dat, které je pro tuto úlohu možné použít, popisuje, které informace z nich lze získat a jaká jsou jejich omezení.

Platformy pro zpracování velkých dat

Systém je koncipován tak, aby jej bylo možné škálovat pro použití v sítích s takřka libovolným provozem. Je tedy nutné, aby byl navržen jako distribuovaný systém. Třetí kapitola představuje problém zpracování tzv. „velkých dat“ a požadavky, které se při něm objevují. Popisuje několik technologií, které je k tomu možné použít. Podrobněji představuje platformu Apache Kafka, která byla pro implementaci využita.

Komponenta pro sběr dat

Klíčovou komponentou systému je sběrač, který zajišťuje sběr dat z různých externích zdrojů. Pro každé vstupní doménové jméno sběrač provádí sken DNS záznamů, získává informace o registraci domény pomocí protokolů RDAP nebo WHOIS, pokouší se připojit na webové servery za účelem získání užitečných dat a certifikátů z inicializace TLS spojení. Sběrač pracuje i s IP adresami, které pro doménové jméno získá z DNS záznamů. Ke každé

se pokouší získat data o autonomním systému, geolokaci a reputaci podle systému CESNET NERD. Zároveň na každou IP adresu vyšle datagram s ICMP zprávou typu Echo a očekává odpověď, přičemž měří dobu odezvy.

Sběrná jednotka je implementována jako modulární systém, který umožňuje snadné přidávání nových zdrojů dat. Každý zdroj dat je reprezentován samostatným kolektorem, který je zodpovědný za komunikaci s příslušným zdrojem, získávání dat a předávání do zbytku systému v definovaném formátu. Kolektory jsou implementovány v jazycích Python a Java, což umožňuje využití specifických knihoven a nástrojů pro jednotlivé typy dat.

Komponenta pro extrakci příznaků

Komponenta pro extrakci příznaků transformuje nasbíraná data do podoby vhodné pro strojové učení. Provádí předzpracování a normalizaci dat, pomocí různých metod počítá 176 různých příznaků, které byly navrženy týmem DomainRadar nebo převzaty z odkazované literatury. Jednotka pro extrakci příznaků je navržena tak, aby byla jednoduše rozšiřitelná. To umožňuje snadné přidávání nových příznaků a úpravu stávajících metod extrakce, což je klíčové pro další vývoj systému DomainRadar.

Dosažené výsledky

Výsledky této práce ukazují, že navržený systém je schopen efektivně a rychle sbírat a zpracovávat velké objemy dat o doménových jménech v reálném čase. Díky použití distribuovaného zpracování na platformě Apache Kafka systém dosahuje vysoké propustnosti. Experimenty prokázaly významný nárůst propustnosti při spolupráci více instancí systému, což umožňuje škálování v závislosti na potřebách jednotlivých případů nasazení.

Systém nasbíral data z osmi externích zdrojů pro 400 000 doménových jmen přibližně za 4 hodiny, čímž dosáhl průměrné propustnosti 28 doménových jmen za sekundu. Nasazení v síti CESNET potvrdilo schopnost systému stabilně zpracovávat data v reálném čase. Vzhledem k technickým problémům na straně CESNET do systému přicházela doménová jména ve významně omezené míře, v jednom z testů byl proto provoz uměle navýšen a systém zde bez obtíží zpracovával průměrně 9,56 doménových jmen za sekundu, čímž naplňuje odhadnuté požadavky.

V závěru práce jsou identifikovány možnosti dalšího vývoje. V budoucnu bude vhodné podrobněji analyzovat využití systémových prostředků jednotlivými komponentami a optimalizovat práci s pamětí. Podrobnějšímu zkoumání by mělo být podrobeno chování serverů poskytujících data o registraci domén pomocí protokolu RDAP, které omezují frekvenci přístupu a mohou tak zpomalovat sběr dat. Navzdory těmto výzvám je systém nyní připraven pro další vývoj, rozšiřování a nasazení v produkčním prostředí.

Effective Large-scale Collection of Information Related to Domain Names

Declaration

I hereby declare that this Masters's thesis was prepared as an original work by the author under the supervision of Ing. Radek Hranický, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Ondřej Ondryáš
30th July 2024

Acknowledgements

I would like to thank my supervisor, Ing. Radek Hranický, Ph.D., for his guidance, advice and the motivation he provided to me (not only) for this work. To a *not insignificant* degree, he is to be thanked for my interest in the academic career – even though it certainly contributed to me spending another summer working on a thesis. Many thanks go to other colleagues from the DomainRadar group for valuable insights and discussions. I also thank the CESNET association for providing data on resolved domain names in their network and the testing environment for the system (an honourable mention goes to V. Bartoš, who saved the day after I managed to break the system before even starting the experiments).

Děkuji svým drahým rodičům za všechnu podporu, kterou mi poskytovali nejen při studiu. Velký dík patří Lukáškovi, Peťulce, Elišce, Denisovi, Terezce, Janči, Džejničce a Alexíkovi, Jakubovi, Adrianě a Toasterovi, Patrikovi, Anežce, Domči, Tondovi, Helči, Míši a Dandovi, Kotymu, Anetce, Lukášovi (z Rudice!), Julii, Jindrovi, Toothpickovi, Jardovi, Kiznohovi, Kukimu, Blážovi. . . Každý z vás pro mě v těch uplynulých pěti letech znamenal něco zcela jiného, vždy něco důležitého – a ač to tady vypadá jen jako zvláštní hromádka jmen, jsou za nimi osoby, na které nikdy nezapomenu. Jag älskar oss.

A já stojím před branou, otočen zády k ní, přede mnou spousta nových dní.

Contents

List of Figures

List of Abbreviations

API	Application Programming Interface
AS	Autonomous System
ASN	Autonomous System Number
CA	Certification Authority
CLI	Command-Line Interface
CPC	Confluent Parallel Consumer
DAG	Directed Acyclic Graph
DBMS	Database Management System
DDL	Data Definition Language (subset of SQL)
DGA	Domain Generation Algorithm
DN	Domain Name
DNS	Domain Name System
DSL	Domain-Specific Language
eTLD	Public Suffix (effective TLD)
IANA	Internet Assigned Numbers Authority
ICANN	Internet Corporation for Assigned Names and Numbers
ICMP	Internet Control Message Protocol
ID	Identifier
IP	Internet Protocol
JDBC	Java Database Connectivity
JSON	JavaScript Object Notation
KC	Kafka Connect
KS	Kafka Streams
ML	Machine Learning
NERD	CESNET's Network Entity Reputation Database
OS	Operating System
POM	Project Object Model (XML format for Java Maven projects)
RDAP	Registration Data Access Protocol
RIR	Regional Internet Registry
RR	(DNS) Resource Record
RRtype	(DNS) Resource Record type
RTT	Round-Trip Time
SLD	Second-Level Domain
SNI	Server Name Indication
SSL	Secure Sockets Layer
TLS	Transport Layer Security
TCP	Transmission Control Protocol
TLD	Top-Level Domain
TOML	Tom's Obvious Minimal Language (configuration file format)
TTL	Time To Live
UDP	User Datagram Protocol
UI	User Interface
UTC	Coordinated Universal Time

Chapter 1

Introduction

The internet, being an integral part of most people's lives, is naturally a seedbed of malicious and criminal behaviour targeted both at individuals and institutions. Attackers often try to gain access to users' online accounts to steal money or gather sensitive information. Some attackers employ *phishing*, where they trick someone into giving their credentials directly to them, for example, by convincingly imitating a legitimate service. Others attempt to install malicious software – *malware* – on the users' devices. This software may track the user, send their data to the attacker or convert the device into a puppet following the attacker's commands and carrying out further harmful actions, such as distributed attacks.

Although the success of such attacks comes primarily from users' unawareness or recklessness, much research has been done on preventing them by technical means. Numerous approaches have been proposed based on various levels of network traffic analysis, for example, by detecting patterns in the network flows or inside the actual communication. While promising results have been demonstrated with methods that detect malicious content from the structure of web pages or URLs [37, 47], their usability is seriously limited by the prevalence of encrypted communication. According to Firefox Telemetry, over 80% of web pages are being loaded using HTTPS worldwide [72]. Without the possession of the encryption keys, it is computationally infeasible to access the transferred data, so threat detection methods independent on the contents of end-to-end communication are needed.

One promising approach of evaluating online risks is based on the analysis of *domain names*, human-readable identifiers of internet servers and other nodes. When using online services, client stations must translate the names to internet addresses using the Domain Name System (DNS). Although several DNS encryption mechanisms exist, their adoption rate is low and does not grow significantly [48], so the DNS queries are usually transmitted unencrypted. Thus, they may be captured at various network nodes and analysed.

While the lexical properties of domain names can provide some threat indicators, enriching them with information from additional sources significantly enhances threat assessment. Monitoring software can determine when and by whom the domain was registered, or it can query the DNS to collect data on how the domain is configured and to what addresses it points. It can evaluate different characteristics of addresses, such as their geographic location, affiliation with autonomous systems, or the presence of a secure HTTP service. By collecting such data for large amounts of domains that are a priori known to be harmless and for domains known to be malign, a machine learning (ML) model can be trained to assess the "badness" of a previously unseen domain, leveraging these diverse characteristics for accurate threat detection.

Contributions

This thesis is a part of a research project at BUT FIT called Flow-Based Encrypted Traffic Analysis (FETA). One of its goals is to develop an ML-based system for domain name threat detection called DomainRadar. Its inputs are domain names captured by probes at the perimeter of a secured network. The system collects data related to the input domain names from various sources, extracts useful features from these data, and passes them to fine-tuned classification models that decide whether the domain seems malicious. The system is meant to be deployed in environments with large throughput, so it must be able to process large amounts of data in a reasonable time.

The system comprises several components: a *loader* for loading and filtering the input domain names, a *collector* for gathering the external data, the *feature extractor* that processes the data, and the *classification subsystem*. This thesis presents the system's overall architecture, designed in cooperation with the DomainRadar team, and proposes the specifics of its realisation.

The main focus of the thesis is the system's "data processing pipeline", *i.e.* the collection and feature extraction components, as well as the data exchange facilities for their interaction with the rest of the system. It presents the design and implementation of these components prepared for the integration with DomainRadar. Additionally, they can be used as a standalone tool, for example, for building new large training datasets.

The implementation leverages distributed computation based on the Apache Kafka platform to achieve the required throughput. Kafka provides scalability as well as fault tolerance. As the system is a matter of ongoing research, the pipeline is designed to be modular and extensible so that new data sources or features can be easily added.

The data processing pipeline was evaluated on a sample of 400,000 domain names sourced from real-life traffic. Its size corresponds to the expected daily throughput of the system. Even without fine-tuning, the system processed all the data in slightly over 4 hours, leaving plenty of room for larger throughputs. Then, it was deployed in the CESNET academic network, where it successfully processed data at a rate of 9.54 domain names per second.

Structure of the Thesis

Chapter 2 presents preceding work on threat detection based on domain names, focusing on approaches and features used in past research. It also outlines the current state of the FETA DomainRadar research. Chapter 3 discusses the problem of large-scale data processing in general and introduces the Apache Kafka platform on which the system is built. Chapter 4 discusses the various sources of domain-related information and data, their limits, and the pitfalls of their usage. It also lists the features that can be extracted from these data and used in the ML models based on literature and the DomainRadar research. Chapter 5 introduces the high-level design of the DomainRadar system, evaluates the expected throughput and uses these findings to specify the requirements for the data processing pipeline. Chapter 6 presents the design of the data processing pipeline, including the data flow, the structure of the data, and the processing steps. Chapter 7 describes the implementation of the pipeline and gives a brief overview of the included deployment facilities. Chapter 8 reports the comprehensive experiments conducted to evaluate the system's performance and resource usage under different configurations. Finally, Chapter 9 summarises the results and outlines the future work.

Chapter 2

Malicious Traffic Detection Based on Domains and Related Sources

Several types of content are generally recognised as malicious, even if no single definition exists. Naturally, there is also a large variety of related research, in terms of the types of examined malicious content and the sources used when building datasets about it [126]. Various forms of online threats, such as phishing, malware, and botnets, pose unique challenges and require different detection strategies.

Phishing

Phishing is a prevalent form of cyber attack where individuals are deceived into revealing sensitive information by mimicking legitimate websites or entities. Attackers use domain names closely resembling trusted ones to manipulate victims into believing they interact with a legitimate source. These attempts range from simple mimicry to advanced techniques like spear-phishing [76], targeting specific groups or individuals within organisations, and whaling [102], which aims at high-level executives with personalised and well-crafted messages. These days, attackers often employ sophisticated social engineering techniques that use personal information from social networks to create convincing messages and fake websites [128]. The rise of mobile platforms has led to smishing [82], where attackers send SMS messages with malicious links to download malware or fill in credentials to fake forms, exploiting the informal and immediate nature of text messaging to catch recipients off guard.

Malware

Websites hosting various types of malware represent a significant category of malicious internet content. These sites lure users into downloading harmful software through deceptive links, with some disguising their domain names to mimic legitimate websites. Some websites are infected with drive-by-download exploits, which are malicious scripts that compromise the browser or its plugins by exploiting vulnerabilities upon user access [22]; others may host or point to binary files containing malicious content.

Botnets and DGAs

Botnets are networks of infected computers controlled by cybercriminals for malicious activities such as distributed denial of service (DDoS) attacks, spamming, and data theft. These networks often rely on domain names for their command and control (C&C) infrastructure, with domains registered in bulk and rapidly changing to avoid detection. Domain generation algorithms (DGAs) are used to create a large number of domain names for botnet infrastructure, which are then used to establish communication channels between infected devices and the C&C servers. This technique, called domain fluxing [124], enables attackers to hide control servers and quickly shift their infrastructure, posing a significant challenge to traditional domain blocklisting approaches due to their unpredictability and volume [122].

2.1 Approaches to Malicious Domain Name Detection

Methods for the detection of malicious domain names in traffic evolved from basic blocklists to using various machine learning approaches. Blocklists (also known as blacklists or denylists) have been a traditional method for the prevention of malicious traffic. However, they have shown limitations due to their static nature. Attackers frequently change domain names, rendering blocklists ineffective over time. Even so, blocklists remain helpful for fast detection of malicious content and are used as components of more complex systems. They are also often used as data sources for machine learning approaches [6, 126].

Machine learning offers a more dynamic approach. Feature-based learning involves extracting and analysing specific characteristics (features) of domain names to determine if they are malicious. A wide variety of features may be used (as demonstrated in Section 2.2). The method relies on the premise that malicious domains often have distinguishable patterns or anomalies in their naming compared to benign ones. Machine learning algorithms use these features to classify and predict the nature of the domains, thereby aiding in the detection of potential cyber threats.

Deep learning relies on layered neural networks to learn and identify patterns indicative of malicious content without prior explicit feature extraction. Although deep learning can potentially offer more robust detection by uncovering complex patterns, it requires substantial amounts of high-quality annotated data and computational resources [39].

Regardless of the specific approach, ML-based methods differ in the character of the inputs they use for classification. Some authors focus on examining the domain names only, while others analyse the patterns in DNS requests and responses. They consider various aspects, such as the frequency of DNS requests, the diversity of IP addresses associated with a domain, and unusual query patterns [6, 126].

2.2 Previously Studied Features for Malicious Domain Name Detection

Suitable data markers – features – are needed to search for malicious patterns in domain names. Many features have been tried to classify malicious domain names using feature-based supervised learning. Some authors successfully employed features based on the lexical properties of the names, which do not require collecting any external data. Others enriched the datasets using various sources: DNS records, RDAP or WHOIS registration information, or IP address information, such as its placement in autonomous systems or geographic location [126]. This section gives a brief overview of some of the features identified in related work. A comparative analysis of different types of features can also be found in a study by Singh and Goyal [115].

Lexical features

Certain characteristics that distinguish malicious DNs from benign ones can be observed solely from the lexical properties of the names themselves. Such an approach may be particularly successful in detecting various DGA families. Many authors demonstrate great results when using lexical features of URLs to classify both phishing and malware online resources. However, preliminary results from the FETA project show that the lexical characteristics of domain names can also help distinguish phishing and malware sites.

Drichel et al. [42] studied 136 different lexical features for DGA classification gathered or adapted from several previous works. They divided the features into three categories: linguistic, structural, and statistical. The first is based on the presence or count of specific linguistic patterns. Structural features capture properties such as the length of different parts of the name. The last category is based on statistical characteristics, such as the frequency distribution of various n-grams. For example, some of the features they used include:

- the number of occurrences of each letter in the alphabet,
- the ratio of lengths of consecutive digit strings/repeating character strings to the total length,
- the consonants to vowels ratio,
- the maximum length of consecutive consonant/vowel/digit strings,
- the sum of digits in the subdomain,
- the proportion of disagreeing characters of the concatenated subdomains and its inverse,
- the sum of the characters interpreted as base 36 digits.

The publicly available source code for feature extraction[1] describes the complete set of features. Their best-performing model uses 76 of these features, achieving results that consider a success, considering that devices infected with DGA-based malware will typically try to access many of these domains, ultimately leading to successful detection.

The problem of purely lexical-based detection has also been previously studied in the context of phishing or malware. Much research has focused on classifying whole URLs where the domain name is only a part of the input. Blum et al. [18] were likely the first to classify URLs in this way [28], employing a bag-of-words model in which they split the URL by specific delimiters and studied the occurrences of unique values. A similar approach was included in the work of Lin et al. [75], who compared the performance of two groups of features: lexical features that use the bag-of-words model and *descriptive* features that describe some statistical characteristics of URLs[2], such as:

- length, longest word length,
- letter/digit/symbol count,
- alphabet entropy (based on estimated probabilities of letters occurring in the URL),

[1]Feature extraction used in [42]: `https://gitlab.com/rwth-itsec/explain/-/tree/master/explain/base/features/examples?ref_type=heads`

[2]This is more in line with what "lexical features" refers to in this work.

- ratio of digits to total length,
- character continuity rate (the length of continuous sequences of letters, digits, and symbols within the domain name, divided by the total domain length).

Some papers have attempted to tackle the problem using natural language processing (NLP). For example, Buber et al. [19] used features based on certain common keywords, brand names, and strings that were similar but not equal to those. However, Hamroun et al. [50] argue that NLP-based methods do not solve the problem effectively.

Lately, some authors have also experimented with lexical classification of malware and phishing websites based on domain names only. Zhao et al. [125] proposed a method based solely on n-gram analysis. It works by dividing the entire domain names into multiple substrings and deeply analysing them in terms of lexical composition and structure. Cersosimo and Lara [23] used the more common approach of extracting features such as length, vowel/consonant/digit ratio, or Shannon entropy of the name. They also introduce a “meaning ratio” feature based on a wordlist that helps determine if a domain name makes sense for a human.

The overview given above only describes approaches that solely used lexical features. However, they have often been combined with other features, which are further discussed below.

DNS-based features

Many authors have used features extracted from the DNS. Some of them have focused on scanning the DNS traffic itself, observing data from subsequent resolutions of a particular domain name by a client, and others have used DNS records to enrich the datasets or combined these approaches.

Perdisci et al. [101] focused on the detection of flux networks which are related to botnets. They divided the requests into day-long epochs, then clustered the DNS queries by certain criteria and used features like:

- the total number of resolved IPs,
- the average TTL of A resource records,
- the numbers of domains sharing an IP,
- IP diversity – the normalised entropy of the /16 network prefixes,
- IP growth ratio – the average number of new IP addresses in each query.

Features based on the TTL (typically the mean value and standard deviation across DNS records) were also used in [49, 114]. Bilge et al. [17] claimed that malicious domains exhibit more scattered usage of TTL values, so they proposed percentages of TTLs falling into ranges [0,1), [1,10), [10,100), [100,300), [300,900) and $[900,\infty)$ as features. They also stated that the total number of different TTL values tends to be significantly higher in malicious domains. However, Prieto et al. [105] argue against using such features, claiming that low TTL values are also found in benign domains.

Hajaj et al. [49] also introduced a feature based on the number of DNS record changes. Kuyama et al. [70] evaluated the distributions of occurrences of the different record types, selecting the number of NS records and the number of MX records as features. Liu et al. [77] used:

- the number of distinct A records,
- the entropy of domain name,
- the number of distinct NS records,
- similarity of NS domain names (the average value of edit distances between every pair of nameserver domain names).

Features based on domain registration data

While some authors argue that using WHOIS data may limit availability or present bottlenecks to detection systems [18, 37, 41], others have proposed solutions that employ features based on various information on domain registration. Chatterjee et al. [26], Hajaj et al. [49] and Shi et al. [114] have all used some of the possible time-based indicators:

- lifetime – the number of days between the expiration and creation dates [49, 114],
- active time – the number of days between the last update date and the creation date [49, 113],
- age – the number of days between the current date and the creation date [26, 113],
- age left – the number of days between the expiration date and the current date [113].

Sadique et al. also used features based on other registration information, such as the administrative or registrar contact, but they do not give a complete list of features. Intriguingly, registration data were not used in much research.

Features based on TLS handshakes

The TLS protocol is built on top of a transport protocol such as TCP or QUIC. The data are transferred in the form of TLS records, where one of the first records in the flow is the "Client Hello" type (from the client to the server), to which the server responds with a "Server Hello" record. In most settings, these records are not encrypted and contain vast configuration data of the flow, such as ciphersuites. In TLS 1.2, the server sends a "Server Certificate" record that contains an X.509v3 certificate chain used for authentication in most key exchange methods [40, Sec. 7.4.2].

The data extracted from TLS handshakes were used in some research that analysed the whole traffic flows. In these scenarios, the systems obtain a detailed view of the character of the communication. Some features have been extracted by analysing the actual payloads of the TLS communication. For example, Barut et al. [16] used the number of ciphersuites, the number of TLS extensions, the length of TLS key exchange, or the presence of certain TLS features advertised by the server. Anderson et al. [4] included the ciphersuite selected by the server, supported extensions, number of certificates, number of Subject Alternative Names in the leaf certificate, validity in days, and a flag on whether the certificate was self-signed.

Torroledo et al. [121] focused on using the server's leaf certificate to detect malware and phishing content. They devised 40 features consisting mainly of boolean indicators of the presence of certain fields and lexical-statistical characteristics of strings found in the certificate. Drichel et al. [41] employed similar features.

2.3 Project FETA and Goals of the DomainRadar Group

The purpose of the Flow-based Encrypted Traffic Analysis (FETA) project is to develop new technologies and tools that will enable the monitoring of encrypted communications and the identification of attacks on network infrastructure [118]. As a part of the project, the DomainRadar group aims to develop an ML-based system for the detection of malware, phishing, and DGA domain names. This software will be able to read a stream of domain names gathered by scanning DNS queries in a network, collect the required data, and pass it to custom fine-tuned models to determine the domain name's maliciousness level. The system is intended to be deployed, among others, in a large academic network that exhibits around 2.3 million unique domain names per day of regular traffic.

One of the primary goals of the DomainRadar group is to develop ML models that will classify a domain name into one of the maliciousness categories. In the final product, the user should be presented with an overview of the detected potential threats where each domain name receives a *certainty score* for being a phishing domain, a malware domain, or a DGA domain, together with an explanation for this decision.

In time of writing, the group has already crafted several classifiers that show promising results. However, all this work has been done on a static set of data collected by a manually operated prototype tool over several weeks. Now, a solid data collection and processing solution without the need for frequent user interventions is needed to put the classifiers into practice, which is what this thesis aims to provide.

Initial experiments

In [20], Bučko presented a binary DGA classifier with an F-1 score of 0.980, and a multiclass classifier with an average F-1 score of 0.8619, which is comparable to or exceeds the performance of the results demonstrated in related work. The team now continues to enhance these results further, mainly by trying other classification algorithms or working with more extensive and diverse datasets.

In [57], Horák developed data collection software and experimented with the XGBoost classifier, an effective system for tree boosting that is usable in classification [27]. He demonstrated promising results in the binary classification of phishing domain names by achieving an F-1 score of 0.9422.

Large datasets and related experiments

We studied various sources of domain names to assemble representative datasets of benign and phishing domains. We were presented with two challenges. First, it was necessary to collect high-quality *ground truth*, lists of unquestionably benign and phishing domains. Second, each domain name was to be enriched with data from related sources (similar to those discussed in Section 2.2).

We used the "Top 1 million queried domains" lists from the Umbrella Popularity List [31] as the basis for the benign set. We filtered the set by a process previously used by Rahbarinia et al. [109] to ensure benignity, keeping only those domain names that appeared in the top list consistently over a year. This yielded a list of 432,572 domain names. The phishing domains were obtained from OpenPhish, an automated platform for phishing intelligence [96], and PhishTank, a collaborative clearing house for data and information on phishing [30]. Both platforms rigorously validate reported domains. The resulting dataset was published on Zenodo [58].

Since publication, we have worked on extending the phishing dataset and on further verification of the records to ensure their quality. We now have data for almost 90 thousand phishing domain names, and we are currently processing data from the CESNET network to assemble another dataset that better captures real-life traffic. At the same time, we are employing VirusTotal, a malware scanning system [29], to assess whether our domain name lists are correctly labelled. In our latest experiments, we worked with a dataset of 110,311 verified phishing domain names.

In the work preceding this thesis, related research was studied to determine more features previously used in malicious domain name classification. Horák's collector was modified to acquire more detailed data. Together with the rest of the team, we also developed novel features that can be extracted from these data. After finishing the dataset and coming up with an extensive feature vector, we compared several different classification methods such as Logistic Regression[3], Support Vector Machines[4], Random Forests[5], or various gradient-boosted algorithms such as AdaBoost[6], XGBoost[7], and LightGBM[8]. Preliminary results, presented in [59], suggest that our approach could be used in real traffic. However, more work is yet to be done to achieve even lower false positive rates, which are critical for practical usage.

Current work

Currently, the team is examining the possibilities of using other classification methods, such as deep neural networks. It is also planned to assess various multi-classifier setups. So far, all experiments have been conducted using a complete feature vector. However, "smaller" classifiers could also give good enough decisions based only on a subset of the features, allowing for a higher level of parallelism. In the upcoming months, we will also shift our focus towards malware domains, employing the findings from phishing detection.

At the same time, we are working on putting the classifiers into practice and creating the actual production-ready DomainRadar software. This thesis is a significant part of this effort, designing and implementing the data collection and processing pipeline. Other team members work on the data loading component, the classifier module, and the user interface.

[3]LLR: https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
[4]SVM Classification: https://scikit-learn.org/stable/modules/svm.html#svm-classification
[5]Random Forests: https://scikit-learn.org/stable/modules/ensemble.html#forest
[6]AdaBoost: https://scikit-learn.org/stable/modules/ensemble.html#adaboost
[7]XGBoost: https://xgboost.readthedocs.io/en/stable/
[8]LightGBM: https://lightgbm.readthedocs.io/en/stable/

Chapter 3

Parallel and Distributed Approaches to Data Processing

The system designed in this work must reliably collect and process data from various external sources (described in detail in the next chapter). The system must withstand a high regular throughput of over 300,000 domain names per day, as well as possible spikes in traffic, making scalability one of the primary requirements for the used design and underlying technology.

This chapter discusses how large amounts of data can be processed in parallel, focusing primarily on distributed computation. Section 3.1 presents the general problems in processing large amounts of data. Section 3.2 explains the differences between the processing of batches and streams. Section 3.3 presents some existing data processing platforms that tackle the problem in various ways. Section 3.4 introduces the Apache Kafka platform, and Section 3.5 shows how it can be used for streaming data processing using various frameworks.

3.1 Big Data Processing

Large-scale processing of so-called *big data* has become a crucial task in the last decade. People and devices produce massive amounts of data that different industries and interest groups can exploit to gain new knowledge. The so-called three "Vs" often characterise the term [129, p. 102]:

Volume refers to the amount of data that grows constantly and can reach the orders of terabytes or petabytes.

Velocity refers to the speed at which the data are produced (and must be processed).

Variety refers to the different types of data, such as text, images, audio, video, etc.

Other "Vs", such as value, veracity, variability, or volatility, can also be discussed [65, p. 9–11]. Processing of such data is not only limited by the storage capacity but also by the processing capabilities of the computation system – in fact, data are often collected at rates that exceed the ability to make use of it [65, p. 2][129, p. 102]. Unsurprisingly, many software platforms have been established to solve the problem of parallel, scalable, fault-tolerant, resilient data processing and analysis with high throughput and low latency.

Parallelism

Parallel processing generally involves executing multiple calculations concurrently, as opposed to the traditional sequential approach, where tasks are executed one after another. It is crucial for handling large-scale data as it improves efficiency and feasibility. The terms "parallel" or "parallelism" in this context are multifaceted. Parallelism can occur at various levels – from instruction-level parallelism inside a CPU through multi-threading or multi-core computing to multi-processing across a network. Furthermore, many other parameters can differentiate parallel systems – different topologies, synchronisation mechanisms, task scheduling strategies, or processing unit types can be used. One of the most notable distinctions is the memory model: in *shared memory* systems, all processes access a common memory space, while in *distributed memory* systems, they communicate by message passing. Programming models using parallel processing can be based on *data parallelism*, where large datasets are divided into smaller chunks processed simultaneously by the same task, or *task parallelism*, which involves executing different computational tasks simultaneously on the same or different data [129, p. 69, 84].

Distributed computation

According to Kshemkalyani and Singhal [67, Sec. 1.1], a *distributed system* is a collection of mostly autonomous processors (nodes) that cooperate, communicating over a network, to solve a problem that cannot be solved individually. It has no common physical clock (introducing asynchrony) and no shared memory; the nodes are separated physically (and even geographically). The architecture of distributed systems can vary widely, from tightly coupled systems, where nodes work closely together on a specific task, to loosely coupled systems like grid computing, where nodes may be more autonomous and geographically dispersed. Nevertheless, it is clear that distributed systems also conduct a form of parallel computation.

Different motivations may lead to the use of distributed systems. Some applications are inherently distributed, especially when reaching consensus among geographically distant parties is required. However, key motivations are often reliability, performance, and scalability [67, Sec. 1.3]. These systems allow for easy expansion by adding more nodes as the demand for data processing grows. By dividing tasks across multiple nodes, they can process data much faster than a single machine. They also provide enhanced fault tolerance, as the failure of a single node does not compromise the entire system. In fact, they are often designed to utilise less expensive commodity hardware, relying on their fault tolerance mechanisms to mitigate possible errors [38].

General requirements on parallel and distributed systems

Various definitions of requirements on parallel and distributed processing systems can be found in the literature. Based on [67, Chap. 1.8.1] and [129, p. 67–68, 223], a distributed system that processes large amounts of data must address, among others, these issues and design aspects:

Communication It must use appropriate mechanisms for communication among nodes.

Processes It must deal with the management of processes and threads and code migration across nodes.

Naming It must be able to devise robust schemes for names, identifiers and addresses.

Synchronisation Mechanisms for synchronisation or coordination among the processes are essential. They are used not only for mutual exclusion but also for leader election, clock synchronisation, or global state recording.

Data storage It must provide fast access to data that is scalable across the cluster.

Consistency and replication Replication of parts of data is needed to provide fast access and scalability. However, it must be ensured that the replicas are consistent.

Fault tolerance Correct and efficient operation must be ensured even in the presence of faults in links, nodes, or processes. Checkpointing and recovery algorithms should be employed to minimise state loss.

Scalability and modularity Algorithms, data, and services must be as distributed as possible. The system must be able to capably scale up to a large number of nodes to handle a growing amount of work. Adding new nodes to the system must be possible without negatively affecting its operation.

Security Secure channels, access control, and secure group management are needed.

Interface and transparency It must provide an interface that hides the data representation, provides uniform access to resources, makes their location transparent, allows their relocation without changing names, masks the concurrent use of shared resources and does not let the user become aware of any failures.

Efficiency Big data processing requires a significant computation capability. Efficiency means faster speed with respect to the usage of certain resources like memory or the number of nodes.

Latency and throughput Latency is the time between the initiation of a request and the beginning of a response. Throughput is the number of such requests that can be processed per unit of time. Both are especially important in real-time processing.

3.2 Batch vs. Stream Processing

Batch processing and streaming processing represent two distinct paradigms in big data processing. Batch processing involves the collection of data over a defined period, followed by the execution of a comprehensive processing task on these accumulated data. This method is particularly effective for large-scale data analysis where a complete dataset is required for accurate processing [129, p. 69]. On the other hand, stream processing emerged as a response to the need for real-time data processing. This approach supports applications that require immediate ingestion, processing, and data analysis as they are being constantly produced (without the delay inherent in batch processing) [129, p. 69].

MapReduce

A typical example of a batch processing paradigm is the MapReduce model proposed by Dean and Ghemawat in 2004 [38]. The input to a MapReduce job is a batch of input data in the form of a key-value pairs list. The batch is split into independent chunks distributed across the cluster. Each input pair is then processed by a user-defined *Map* function that outputs a list of intermediate key-value pairs. The framework groups them by key and stores the intermediary pairs on local disks. A *partitioning* function decides how the intermediary key space will be distributed across the nodes. The groups are sent (*shuffled*) through the network and passed to *Reduce* functions that produce the final key-value results [38].

One of the key ideas of MapReduce was that data should be moved around the cluster as little as possible. The data are distributed in chunks and replicas across the cluster, and the MapReduce master plans to schedule the Map tasks on machines that contain the chunk they process. MapReduce excels in its simplicity and scalability. It was designed to run on commodity machines that are easily replaceable and can be added to the cluster as needed. Additionally, it enables reaching the necessary fault tolerance: if a node or its task fails, the system may reschedule it on another machine [38].

However, this design is less suitable for real-time, event-based applications with continuous and unbounded input data streams. First, MapReduce materialises the intermediary values into a local file before subsequent processing can take place [38]. Second, the Reduce phase may only start when the entire batch is ready – this inherently cannot ever happen with unbounded, infinite input data streams.

Dataflow models

Streaming systems are often based on the *dataflow* approach, in which the computation is expressed as a directed acyclic graph (DAG) where nodes represent operations and edges represent data dependencies [129, p. 237]. A system implementing this model plans the actual execution of the operations and the distribution of data across them. Dataflow may exploit both task and data parallelism. The former is achieved by executing several independent operator tasks on a single node. The latter is achieved by partitioning the data into chunks that can be processed independently [60, Chap. 2].

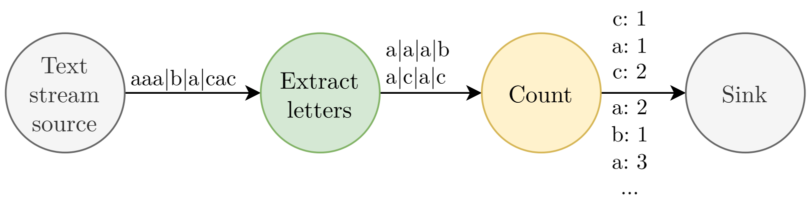
Figure 3.1 shows a simple dataflow program that accepts a stream of words, splits them into characters, and counts the total number of their occurrences. The second phase is a *stateful* operation – it keeps a record of the total number of occurrences per key and outputs updated values as they come, providing real-time analytics on the stream. While Figure 3.1a shows the program as defined by the programmer, Figure 3.1b suggests how the platform may transform the program into a graph of tasks that it deploys on the nodes. Note that the whole target pipeline may still run on a single node; the job planner would optimise the execution plan based on various parameters, including the available resources.

Micro-batching

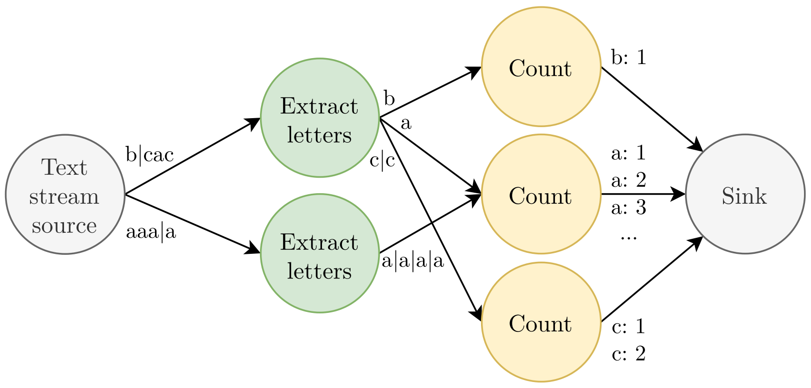
Another way of processing real-time streams is *micro-batching*, a hybrid approach aiming to bridge the gap between the need for real-time processing and the efficiency of batch operations. This model collects data in small, discrete intervals – batches. Each of them is processed as it arrives. This method allows for near-real-time data processing. The downsides of this approach include discretisation latency, possible under-utilisation, and restricted expressibility in contrast to dataflow systems [129, p. 238].

Message delivery semantics

One of the most challenging concepts in streaming data processing is achieving the *exactly-once* semantics. [129, p. 240–245] elaborates on the topic, noting that processing guarantees



(a) A logical dataflow graph. The nodes represent operators.



(b) A possible realisation of the dataflow program. Here, nodes represent actual tasks running on a node.

Figure 3.1: A simple dataflow program and a scheme of its possible physical realisation. The green nodes are stateless transformations, and the yellow nodes are stateful transformations, aggregating records as they come. Based on [60, Fig. 2-1, 2-2].

refer to the state of an application. A system ensuring exactly-one semantics verifies that any application it runs will consume its input without record losses, and all declared internal states will be updated once per record, even in the presence of failures. However, it does not guarantee that the application's output will be consistent under failures.

To achieve exactly-once semantics, it should hold that:

1. *[Guaranteed processing]* All records in a task's input dependencies are eventually delivered to the task and fully processed, *i.e.* the task receives the input record, updates its state, and produces output accordingly.

2. *[Consistent state updates]* Each input record leads to exactly one state update.

Sometimes, a less strict semantics is required. The *at-least-once* semantics guarantee satisfies Property 1, *i.e.* each record will be processed at least once, but it may be processed multiple times. This is easier to achieve, as the system can reprocess the record if it is unsure whether it was processed before.

3.3 Examples of Distributed Data Processing Frameworks

There are many distributed data processing frameworks available today. Some of them are general-purpose, while others are designed for specific tasks. This section briefly introduces some popular frameworks used for big data processing.

Apache Spark

Apache Spark is an open-source analytics engine for large-scale data processing. Spark overcomes some of the limitations of the MapReduce model: it takes advantage of in-memory processing, offering improved speed; it uses an advanced DAG-based execution engine and a more flexible job execution planning. It abstracts data using Resilient Distributed Datasets (RDDs), immutable data collections distributed across a cluster that support parallel operations like mapping, filtering, and reduction [79, Chap. 1].

Spark supports both batch and streaming data processing. Initially, it used the Discretized Stream (DStream) abstraction for streaming, which was internally implemented using sequences of RDDs [79, Chap. 16]. Spark 2.0 introduced Structured Streaming, which treats streams as continuously appended tables. This model unifies batch and streaming workloads and allows the expression of streaming computations as standard batch queries [79, Chap. 2].

Spark's ecosystem includes a distributed SQL engine, machine learning libraries (MLlib), and graph processing capabilities (GraphX). Spark is accessible through Java, Scala, Python, and R APIs, enhancing its versatility and making it suitable for various data processing tasks [79, Chap. 1].

Apache Flink

Apache Flink is another open-source platform for scalable batch and stream data processing. At its core, it is a dataflow engine that provides various abstraction layers, including a batch mode. It allows the processing of events from one or more streams and provides a consistent, fault-tolerant state [60, Chap. 1]. Flink's primary abstraction is the DataStream API, which offers transformations on data streams sourced from various inputs and delivered to sinks. The API supports a rich set of transformation operators for mapping, filtering, joining, and windowing streams. It also allows for iterative computations and stateful operations, ensuring fault tolerance with checkpoints and savepoints [7].

Flink also offers the Table API, which provides a relational view of stream data and can be accessed directly via code or SQL syntax. This API is built on top of the DataStream API and supports both bounded and unbounded data, allowing seamless integration between the two [60, Chap. 11]. The range of data to include in stateful operations, such as joins and aggregations, is controlled using windows and watermarks that measure progress in event time, providing a framework for working with out-of-order and late data in streams.

Flink is based on Java and uses the Java Virtual Machine (JVM) to run user code. A separate Scala API was available, although it is now due for deprecation [123]. A Python API providing the features of both the DataStream and Table APIs is also available.

Apache Beam

Apache Beam is an open-source data processing framework that provides a unified model for both batch and streaming data processing. Its core aspect is the Dataflow model described by Akidau et al. in [3]. It focuses on bringing a simple expression of parallel computation over event-time-ordered data from unbounded sources, using the concept of windowing and triggers to prevent relying on any notion of completeness. Beam's data abstraction is the Pipeline, which represents a data processing job composed of immutable datasets that can be either bounded or unbounded. It supports fundamental transforms such as grouping by key, combining, flattening, and partitioning, and it offers the creation of custom processing logic using the “Parallel Do” transform. Beam was designed to separate the logical notion of data processing from the actual implementation. The pipelines can be defined using multiple programming languages, the most mature of which are Java and Python. Beam translates them to tasks for Apache Flink, Apache Spark, Google Cloud Dataflow, Apache Samza, Apache Nemo, or Twister2.

Akka and Akka Streams

Akka is an open-source toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications on the JVM. It leverages the *Actor Model*, where actors are independent entities interacting through asynchronous message passing [129, p. 43]. This model simplifies scalable and fault-tolerant system construction by eliminating shared state and complex threading management, with Akka capable of handling millions of actors simultaneously, providing robust fault-tolerance and built-in clustering for scalability and resilience in distributed environments [74]. Akka Streams, a module within Akka, facilitates non-blocking, asynchronous data stream processing and transformation, focusing on backpressure handling to maintain system stability by preventing fast producers from overwhelming slow consumers [73]. Its declarative API allows developers to compose data flows with pre-defined operators like mapping, filtering, and grouping, enabling sophisticated processing logic.

3.4 Apache Kafka

The system developed in this work will be based on Apache Kafka[1]. It is a distributed event streaming platform designed for high-performance data pipelines, streaming analytics, and data integration[2]. Kafka enables efficient handling of data feeds, acting as a broker between data producers and consumers. It is designed to support fault-tolerant storage and scalable message transfer, allowing it to function as a backbone for real-time processing of data streams. It is a distributed system with a cluster of servers communicating with each other and their clients via TCP [9].

Events, topics and scalability through partitioning

A Kafka message is called an *event* and consists of a key, value, timestamp and optional headers. Except for the timestamp, all these values are opaque for Kafka – it sees them as variable-length binary data (byte strings) [8, Sec. 5.2]. Events are organised in named *topics*. A topic is essentially a persisted event log, so events may be read as often as needed, both as they occur and retrospectively. The position of the event in the log is called the offset.

The topics are *partitioned*: the event log is split into separate partitions that may be distributed across servers in a cluster [9]. Events are typically assigned to a partition based on a hash of the key [8, Sec. 3.1]. This allows Kafka to scale horizontally by adding more servers to the cluster, each of which can handle a subset of the partitions.

Producers publish events to topics and *consumers* read from them. Zero, one, or more producers and consumers may always publish/subscribe to a topic. Multiple consumer instances may form a *consumer group*. Kafka ensures that each topic partition is consumed

[1]Apache Kafka: `https://kafka.apache.org/`

[2]Before reading this section, I encourage the reader to visit `https://www.gentlydownthe.stream/`.

by exactly one consumer per group. An efficient consumer offset management system keeps track of which messages in which topics have been consumed by a consumer (or a group). The concept of events in partitioned topics is demonstrated in Figure 3.2.

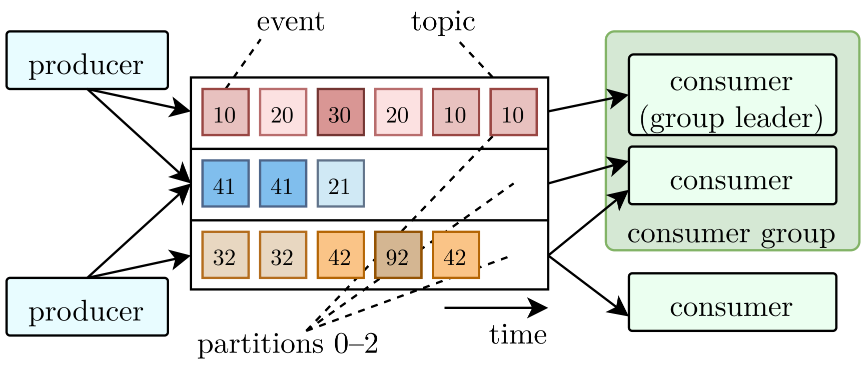


Figure 3.2: A Kafka topic with three partitions. Events (squares) are produced by independent producers. A producer always controls the target partition of an event, but by default, the partition is chosen from the key (numbers in the rectangles) using a partitioning function. In this image, it could be the key value mod 3. Consumers may be standalone or in a group, where the partitions are assigned automatically across the group members.

Through the group rebalancing mechanism, Kafka ensures that all events will be consumed and the load will be evenly distributed, even as consumer instances fail and new ones are added [8, Sec. 4.5, 4.6]. This way, partitioning can be used as the basis for consumer-level parallelism. Increasing the number of partitions makes it possible to scale a data stream, represented by a topic, to be processed by multiple instances of the same consumer in parallel without any extra work.

Kafka can discard events from the log after a configured retention period or keep them indefinitely [8, Sec. 3.1]. Alternatively, the topic may be *compacted* – in this case, Kafka retains only the most recent event for each key, as illustrated in Figure 3.3. The minimum compaction lag provides a lower bound on how long each message will remain in the log. The maximum compaction lag controls the maximum delay before the message becomes eligible for compaction. Kafka's log cleaner compacts a message if the configured "maximum lag" has passed or when the log size exceeds a specified threshold and the configured "minimum lag" has passed. This mechanism helps maintain the latest state per a key, such as the last update of a database record [8, Sec. 4.8].

Message delivery guarantees

Kafka can provide three levels of message delivery guarantees between a producer and a consumer [8, Sec. 4.6]:

- *At most once* – messages may be lost but are never redelivered.
- *At least once* – messages are never lost but may be redelivered.
- *Exactly once* – each message will be delivered exactly once.

The at-least-once semantics are achieved by default. On the producer side, Kafka supports idempotent delivery, preventing duplicates even if the producer retries sending

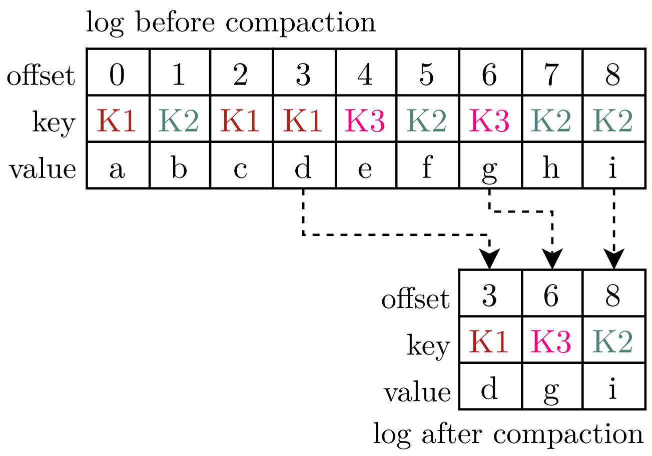


Figure 3.3: Kafka log compaction. Eventually, only the latest event for each key is retained.

messages. On the consumer side, the consumer receives the message, processes it, and then commits the offset. If the consumer crashes before committing the offset, the message will be redelivered. The at-most-once semantics can be achieved by disabling retries on the producer side and committing the offset before processing the message on the consumer side. The exactly-once semantics are achieved with the built-in support for transactions [8, Sec. 4.6].

Clusters, replication and leadership

A Kafka server that stores data is called a *broker*. Kafka may be deployed in an "all-in-one" setup with a single broker, but it is often used in a cluster configuration. A modern Kafka cluster deployment consists of multiple brokers and controllers. The brokers handle data storage and exchange. The controllers manage the metadata for the cluster and participate in the metadata quorum. To manage the cluster, Kafka uses KRaft [8, Sec. 6.10], a consensus algorithm based on Raft [95], which guarantees that the system agrees on a single state despite failures. A server may act as both a broker and a controller, or it can be dedicated to one of these roles [8, Sec. 6.10].

Topic replication in Kafka ensures data durability and availability by creating multiple copies of each partition across different brokers based on global or per-topic settings. The controllers manage the assignment of these replicas, ensuring that each partition has one leader and multiple followers. The leader handles all read and write requests, while followers replicate the data from the leader to maintain consistency. This replication mechanism ensures that if a broker fails, the partitions it was leading can be seamlessly taken over by one of the in-sync replicas, providing fault tolerance [8, Sec. 4.7].

Security

Kafka provides mechanisms for authentication, authorisation, and traffic encryption. For authentication, Kafka offers SSL/TLS (Secure Sockets Layer/Transport Level Security) and SASL (Simple Authentication and Security Layer). SSL/TLS may be used for client-to-broker, broker-to-client and broker-to-broker authentication and encryption of the communication [8, Sec. 7.2, 7.3]. SASL offers multiple authentication mechanisms such as Kerberos

(GSSAPI), PLAIN, SCRAM (Salted Challenge Response Authentication Mechanism) and OAuth for flexible authentication solutions [8, Sec. 7.4].

Authorisation in Kafka is managed through Access Control Lists (ACLs) [8, Sec. 7.5]. ACLs define which principals (identified clients) have permission to perform specific operations. Kafka also provides a pluggable authoriser interface that allows custom authorisation implementations, enabling users to integrate Kafka with their existing security infrastructure.

Kafka Connect

Kafka Connect (KC) is a framework included with Apache Kafka that integrates Kafka with various external systems. It facilitates importing and exporting data to and from Kafka topics through *connectors*. They handle the necessary logic to interact with different data sources or destinations. Connectors exist for various database systems, key-value stores, search indexes, or file systems. Custom connectors can also be developed in Java to support specific use cases. KC simplifies building data pipelines, reducing the need for custom integration code and allowing more effortless data movement between systems [8, Sec. 8.1].

The architecture of KC is designed for distributed and scalable operation. The KC runtime and the connectors can be configured in a distributed mode to split the work into multiple tasks, workers, or both, enabling parallel processing and high throughput. It also provides built-in functionality for monitoring and managing connectors, including automatically restarting failed tasks or changing the runtime configuration through an HTTP REST API [8, Sec. 8.2].

When configuring connectors, users may use the API or define properties in configuration files, including settings for connection details, serialisation formats, and specific connector behaviours. To translate between different data representations, entries from the external systems and from the Kafka topics are first converted to internal structured data types before being converted to the target system's format. Custom converters can be developed in Java to support different data formats or serialisation methods [8, Sec. 8.2].

In addition to data movement, KC provides data transformation capabilities. Transformations can modify individual messages before KC writes them to the target. They can perform tasks such as filtering fields, masking sensitive information, or modifying message structure. They are configured as part of the connector's settings and run in the context of the KC framework, ensuring that data can be adjusted to the needs of the downstream systems without requiring additional processing layers. Custom transformations may again be developed in Java [8, Sec. 8.2].

Role of Kafka in Big Data processing

Kafka's role in the ecosystem of big data technologies is mostly complementary to systems like Apache Spark and Apache Flink [60, Chap. 2b, 9b]. While Kafka excels at data ingestion and dissemination, Spark and Flink are more focused on complex data processing and analytics. Kafka is often used in conjunction with these platforms, where it acts as the aggregation source that collects data from various sources, such as sensors or user activities, and delivers them in a unified way into the processing engines that provide the computation and transformation capabilities. They may then feed the results back to Kafka for storage or further processing by other tools [8, Sec. 1.2].

Kafka in the cloud

Redpanda[3] is a high-performance, Kafka-compatible streaming data platform designed to handle demanding workloads with minimal operational complexity. Unlike Kafka, Redpanda is written in C++ and achieves significant performance improvements, particularly in environments requiring high throughput and low latencies – the vendor claims that it achieves ten times lower tail latencies and six times faster transactions. One of Redpanda's goals is simplifying deployment and management. It provides synchronous replication and in-built data recovery mechanisms, making it a robust option for enterprises seeking to reduce operational overhead while maintaining compatibility with Kafka APIs [110].

Confluent Cloud[4] is a cloud-native Kafka-compatible service built on another custom implementation, the Kora Engine, that eliminates the operational burden of managing Kafka clusters and offers features like auto-scaling, automated updates, and seamless multi-cloud deployments. They also claim that the service offers up to 10 times lower tail latencies than Apache Kafka [33].

Confluent Platform[5] is an enterprise-grade distribution of Apache Kafka, available both for on-premise deployments and as a fully managed service. Developed commercially by the original creators of Kafka, Confluent Platform includes advanced security, management tools, and integration capabilities. The tools provided include ksqlDB for real-time stream processing using SQL, Schema Registry for managing data schemas, and connectors for integrating with various data sources and sinks [34].

3.5 Stream Processing with Kafka

Although Kafka is not a data processing platform at its core, it provides the necessary infrastructure for building stream processing applications even without using heavyweight platforms such as Apache Flink. In simple data processing tasks, one can use Kafka consumers and producers directly to accept data from a topic, process them, and publish them to another topic. This approach lacks the advanced features of stream processing frameworks, such as windowing or cross-data-stream operations. However, it turns out that even complex operations can be implemented using the primitives provided by Kafka.

Mainstream processing platforms, such as Apache Spark and Apache Flink, typically work by providing a runtime environment and infrastructure into which developers submit their code. They require additional infrastructure (*e.g.* a resource manager) and management. The Kafka-based client stream processing libraries choose another approach: they are standalone applications that leverage Kafka's capabilities (such as partitioning, fault tolerance, and group rebalancing). Scaling is achieved by adjusting the partitioning and starting more instances of the applications, for example, by running them in containers and orchestrating with Kubernetes or Docker Swarm.

Kafka Streams[6] (KS) is a Java stream-processing library. It provides a dataflow application model: it represents computation as a DAG where nodes are *stream processors*

[3]Redpanda: https://redpanda.com/
[4]Confluent Cloud: https://www.confluent.io/confluent-cloud/
[5]Confluent Platform: https://www.confluent.io/product/confluent-platform/
[6]Kafka Streams: https://kafka.apache.org/documentation/streams/

and *streams* are the edges. A stream is an abstraction over a Kafka topic. Developers may use the high-level Streams DSL, which includes the common data transformations, and the lower-level Processor API for more complex operations. KS is developed as a part of the Apache Kafka Project [10].

Faust[7] is a Python stream-processing library with capabilities similar to KS. It is inspired by the actor model and based on Python's coroutine-based asynchronous processing[8]. In contrast with KS, its primary interface is not a DSL but a common imperative Python code [112].

Quix Streams[9] is a Python library that uses a table-oriented approach instead. It abstracts the Kafka topics using "Streaming DataFrames" that provide a pandas-like interface, extended with stream processing primitives [108].

Goka[10] is a stream processing library for Go that provides functionality similar to KS. It is based on the concepts of processors, views, and emitters, which define stream processing logic, manage state, and produce new events [78].

All these libraries work essentially by providing a higher-level abstraction over the Kafka consumer and producer, implementing the streaming processing primitives, such as windowing, joins, and aggregations. They achieve this by maintaining the local state in memory or fast local databases (such as RocksDB[11]) and storing it in Kafka topics for fault tolerance.

Duality of streams and tables

All the libraries mentioned above provide the concept of *duality of streams and tables*. Often, practical stream processing use cases require both streams and databases. For example, an e-commerce application may enrich an incoming stream of customer transactions with the latest customer information from a database table. For this reason, stream processing technologies focus on providing first-class support for both. It is often based on a simple relationship between streams and tables (here, a table is interpreted as a collection of key-value pairs) [10]:

- A stream can be viewed as a *change log* of a table: each record in the stream represents a state change (insert, update, delete) to the corresponding key in the table. A stream can be thus turned into a table by replaying the change log from the beginning, possibly aggregating the changes along the way.
- A table can be considered a snapshot of the latest values for each key in a stream at a single point in time, possibly over a given time window. A table can be turned into a stream by iterating over the key-value pairs and emitting a record for each.

Confluent Parallel Consumer

The libraries presented above are tailored for complex stateful and cross-stream operations. However, some data-processing tasks, such as data enrichment or transformation, that

[7]Faust: https://faust-streaming.github.io/faust/

[8]Refer to https://docs.python.org/3/library/asyncio.html for more information on Python's asynchronous I/O.

[9]Quix Streams: https://quix.io/docs/quix-streams/introduction

[10]Goka: https://github.com/lovoo/goka

[11]RocksDB: https://rocksdb.org/

focus on per-item processing do not require these operations. In these cases, achieving high throughput and low latencies is more critical. The Parallel Consumer[12] library by Confluent is a layer over the classic Java-based Kafka consumer that lets the user increase the parallelism of data processing without increasing the number of topic partitions. This improves throughput and latency for certain use cases by reducing the load on the Kafka brokers. It offers an easy way of connecting to other services efficiently via non-blocking I/O without stalling the application. It also provides features for client-side work queues, including message-level acknowledgement and key-based processing. These are great for implementing low-latency task queues, a problem that Apache Kafka does not address on its own [117].

[12]Confluent Parallel Consumer: `https://github.com/confluentinc/parallel-consumer`

Chapter 4

Sourcing Domain-Related Data and Features

Section 2.2 discussed various characteristics of domain names that are useful in classifying malicious domain names. In most cases, these features are derived from related data that must be first collected from external sources. This chapter discusses the practicalities of using these sources, their limits and pitfalls. It indicates some of the features that the DomainRadar team has been experimenting with during its research, and demonstrates how they can be extracted from the data.

4.1 Scanning the DNS

There are many possible features based on the data that the Domain Name System can provide. A domain name designates a domain, that is, a region within the DNS information space. The name consists of a sequence of *labels* separated by dots. Each label represents a node in the DNS hierarchy.

The namespace is divided into *zones* that represent a particular portion of the space. The control over a zone is delegated to a specific entity (individual, organisation) [84, 103]. While a domain is a part of the DNS hierarchy, a zone represents the part of that hierarchy for which a particular nameserver has authority. When a query is made for a specific domain, these authoritative nameservers provide the necessary information to resolve the required record associated with the queried domain name [84].

The information stored in nameservers is organised into *resource records* (RR). Each type of RR contains details about a specific aspect of the domain. A RRset is a collection of these records, typically grouped by their name and type [46]. In the research, the following RR types were used:

SOA (Start of Authority) records hold administrative information about a zone, including details about the zone's primary nameserver, the contact for the zone's administrator, and various timers related to refreshing the data [85].

NS (Name Server) records indicate the authoritative nameservers for the zone [85].

A (Address) records link a domain name to its corresponding host IPv4 address [85].

AAAA records link the domain name to an IPv6 address [120].

CNAME (Canonical Name) records represent aliases for canonical names. They allow a domain to be aliased to another domain, enabling scenarios where multiple domain names resolve to the same host [85].

MX (Mail Exchange) records are essential for email functionality, directing email to the correct mail server for a domain [85].

TXT (Text) records can hold arbitrary descriptive text [85]. These days, they are used for various purposes, including storing verification strings for external vendors, such as when confirming domain ownership. They also play a crucial role in email security, being utilised in frameworks like SPF [66], DKIM [35], and DMARC [68], which are vital in validating email sources and preventing email spoofing.

Caching is crucial in enhancing the system's efficiency by reducing the load on DNS servers and accelerating the domain name resolution process. When a recursive resolver queries a nameserver, the response with the IP addresses and other relevant information is stored in its cache. This information includes a Time to Live (TTL) value, which dictates how long it should be stored before being considered outdated. Subsequent requests for the same domain can be answered directly from this cache, bypassing the need to go through the entire resolution process again [84].

Top-level domains vs public suffixes

The term *top-level domain* (TLD) refers to a domain at the top of the DNS hierarchy. A TLD is subdivided into second-level domains (SLDs); these may be subdivided into third-level domains, and so on [104]. The root of the tree is administered by IANA[1] (now managed by ICANN[2]), which delegates the control over TLDs to other entities [103]. Various types of TLDs exist; the most common are generic TLDs (gTLDs), such as `.com` or `.org`, and country-code TLDs (ccTLDs), such as `.uk` or `.cz`.

The structure of TLDs is often flat: registrants (people, organisations) register their domain directly under a TLD, so they have control over their SLD, and any further structure is up to them. However, especially among the ccTLDs, there is a wide variation in the structure [103]. For example, in the `.uk` TLD, registrants can obtain an SLD directly or register at the third level under the `.co.uk`, `.org.uk` or `.me.uk` SLDs [91]. In the past, registering an SLD directly in the `.uk` TLD was not possible, so many UK websites still run under the `.co.uk` SLD.

The term "public suffix" denotes domain name suffixes under which internet users can (or historically could) register names directly [87]. A public suffix list is available at [87], an initiative of Mozilla, which is kept up-to-date by the online community[3] and the internet registrars.

It is necessary to base some features on the registered domain, *i.e.* the zone managed by the name's owner, not on the input domain name, which may be a subdomain. For example, suppose a name with a certain number of parts is generally likely to be malicious. In that case, UK names would be more often classified as malicious because many have an extra part "by design".

This thesis uses the term *effective TLD* (eTLD) to denote the public suffix of a domain name. Similarly, *effective SLD* (eSLD) denotes the next level after an eTLD; the

[1]The Internet Assigned Numbers Authority: https://www.iana.org/
[2]Internet Corporation for Assigned Names and Numbers: https://www.icann.org/
[3]The public suffix list repository: https://github.com/publicsuffix/list

following level is denoted as e3LD, etc. *Zone domain name* denotes the domain name of the zone of authority in which a domain name resides. For example, consider the domain name `www.site.example.co.uk`: its eTLD (public suffix) is `co.uk`, and its eSLD is `example.co.uk`. The zone domain name is likely also `example.co.uk`. However, if the owner of the eSLD further delegated authority over e3LDs, the zone DN could be `site.example.co.uk`.

The system will build a complete image of the actual configuration associated with the name. Specifically, it will use information on what RR types are associated with the DN, the TTLs of the available RRsets, the contents of A, AAAA, MX, NS, TXT, and CNAME records, whether the domain supports DNSSEC, the zone domain name, and the SOA record of the zone.

Features: The features derived from such data include the number of RRs found for each type, the total number of RR types available, a boolean value indicating whether DNSSEC is available, the mean and standard deviation of TTLs, the distribution of TTL values in terms of certain intervals, the values from the zone's SOA, boolean values indicating the presence of TXT records related to email security, or the number of well-known vendor strings found in TXT records.

Determining the authoritative information

In typical scenarios, the client would typically send the query to a pre-configured recursive resolver, which performs the resolution process, caches the result, and returns it to the client. The TTL parameter of each RRset controls the caching time, though it cannot be taken for granted that the resolver follows it. However, this process is unsuitable when trying to capture the configuration of the name, as the resolver can skew the data. For example, the resolver might shorten the TTL for operational purposes or even completely ignore the original TTL setting [54]. Second, it was observed that the TTL value returned by caching resolvers often carries the value of "time left in the cache" instead of the "maximum time to live" defined in the RR. It does not make sense to characterise a DN with such values.

The system must work with the authoritative data on each DN provided by the authoritative DNS servers for the zone in which the DN resides. The collection process for a DN must thus start by determining its zone domain name and the zone's authoritative nameservers, which can then be queried with the actual target DN. One way would be to implement the iterative DNS resolution process, starting with the root nameservers and following the chain of delegations until the zone's authoritative nameservers are reached. However, this would be unwise because of the practical complexity of the process.

Instead, a standard recursive resolver can be used for finding the zone. Each zone is denoted by a SOA record identified by the zone domain name. The process can thus query the DNS system, using a recursive resolver, for a SOA record of a name, removing the leftmost label until a matching SOA record is found in the *Answer* section. If the domain does not exist, the process is stopped when an eTLD is reached.

The algorithm may be optimised using the *Authority* section of the DNS response. If a matching RRset does not exist, the response should include the SOA record of the nearest known ancestor zone in the *Authority* section. However, the DNS algorithm [84, Sec. 4.3.2] defines a particular change of behaviour in the case of CNAME records: Essentially, if a SOA query is made for a domain name that matches a known CNAME record, the response includes the SOA record of the target of the CNAME, instead of the actual zone in which the input domain resides. However, it is possible to recognise that this has happened by the presence of a record in the *Answer* section. The process is demonstrated in Algorithm 4.1.

Algorithm 4.1: DNS zone/SOA discovery starting from the most specific name

```
Input: domain name $d$
Output: (zone domain name; zone's SOA record) or $\perp$ if not found
1  query a recursive resolver for an SOA record of $d$
2  if the response contains no answer then
3      if the query contains an SOA record in the Authority section then
4          $d' \leftarrow$ the SOA record owner name
5          if $d'$ is an eTLD then
6              return $\perp$
7          else
8              return ($d'$; the SOA record)
9  while $d$ is not an eTLD do
10     query a recursive resolver for an SOA record of $d$
11     if the response contains an SOA answer matching the input then
12         return ($d$; the SOA record)
13     else
14         $d \leftarrow d$ without the leftmost label
15 return $\perp$
```

The SOA record contains the DN of the zone's primary nameserver. Its IP address must also be resolved, which can also be done using the recursive resolver. The resulting IP is the target for the subsequent queries on the examined RR types labelled by the target DN. It is also useful to store the SOA record, as the other values found within may also be used in classification.

The issued queries may ask for NS records instead of the SOA record. These are also mandatory for each zone and contain the names of all the zone's authoritative nameservers. They can be used in case the primary nameserver does not respond.

DNSSEC

DNSSEC is a suite of specifications for securing certain information the DNS provides. It protects the system from attacks such as DNS cache poisoning by providing origin authentication and integrity protection for DNS data, as well as a means of public key distribution [14].

Although DNSSEC intends to enhance security, its presence or absence does not seem to be a particularly good indicator of maliciousness. A domain that implements DNSSEC demonstrates a commitment to security by having used a "good" registrar. However, the lack of DNSSEC does not necessarily mean a domain is malicious, as it is not yet widely deployed (recent estimates show that fewer than 10% of the domain names used for websites are signed [55]). The results of DomainRadar experiments show that DNSSEC-based features do not significantly contribute to maliciousness detection. For this reason, the only related feature will be the presence of the DNSSEC record type in the zone, without checking the chain of trust or the validity of the signatures.

4.2 Sourcing Registration Data using RDAP and WHOIS

WHOIS is a simple internet service that registrars provide to query data on registered names. It usually provides information such as the domain's first registration date, last update date, expiration date, or the registrar and their contact information [62, Sec. 1.4], sometimes the registrant's contact information (although this has been largely limited since the EU's general data protection regulation, GDPR, became effective in 2018 [64]).

The specification of the underlying protocol given in RFC 3912 [36] is quite vague, as it does not define the data model or the format of the responses. ICANN's Registrar Accreditation Agreement (RAA) specifies a "semi-free text format" and requires a unified format for some fields to provide interoperability [62, Sec. 1.3, 1.4]. However, this agreement is binding only for gTLDs – the control over ccTLDs is almost entirely delegated to their managers in the respective countries. For this reason, they may not provide a WHOIS service, or they may omit some data. The informative RFC 7485 [127] shows that only 85% of 124 TLD registries provide the creation date. Furthermore, the protocol lacks internationalisation support and does not implement any security features such as client authentication, making its usage tricky and prone to errors.

A new protocol called *Registration Data Access Protocol* (RDAP) was developed to overcome these issues. The features of the protocol, including a RESTful query interface, a structured response format employing JSON, and security services, were defined in a set of RFCs (7480 to 7484). ICANN also published a detailed profile and implementation guide [61]. Since a 2023 amendment to the RAA, ICANN requires all gTLD registrars to provide RDAP access to their domains and set a "WHOIS Sunset Date", after which they will no longer be required to provide WHOIS service [62, 63]. Once again, this only applies to gTLDs, as ccTLDs are not bound by this agreement. In January 2024, only 28 ccTLDs were registered in IANA's RDAP bootstrap file[4]. On the other hand, over 35% of all domain names are registered in the .com TLD that offers RDAP. Furthermore, the system will first be deployed in a Czech environment, where extensive use of the .cz TLD can be expected, and the manager of this TLD also provides RDAP access.

Features: In addition to registration data, RDAP provides additional information, such as the domain's nameservers or whether the parent zone contains a DNSSEC signature of the domain's key. Also, the research suggests that the registrar might be a strong discriminative feature. The system will use features such as the total registration period, the number of days elapsed since registration and the last update, a boolean value indicating that the zone is DNSSEC-signed, a hash and lexical characteristics of the registrar's name, or a hash and lexical characteristics of other entities found in the data, such as registrant or abuse contact.

Rate limiting and ways of mitigation

Whatever of the two services is used, they both present a severe challenge: rate limiting. When building a training dataset, it was observed that both WHOIS and RDAP servers started to block requests from the collector's IP address after a certain number of queries had been made. The operators of the servers are in total control of this behaviour – it is not specified in any way for the WHOIS servers; for RDAP, [88, Sec. 5.5] only mentions that they may choose to implement rate limiting and should respond with the 429 HTTP

[4]RDAP Bootstrap Service Registry for Domain Name Space: https://data.iana.org/rdap/dns.json

status code (though they are free to return another response code, not to reveal that the address has been rate limited). A “brute-force” solution to the problem is to use a large (enough) number of source IP addresses and distribute the queries among them. However, this is likely not feasible in practice.

For some of the TLDs that support RDAP, there is a possibility of arranging a special agreement with the registry that would allow for a higher rate limit, especially when used in practice. Because RDAP provides standardised means of authentication, the registry could identify the collector by using an authentication token. While not a general solution, it could help mitigate the problem for TLDs that are observed more frequently.

Other commercial services also exist that allow access to registration data in a structured and unified way, such as Bulk Whois Api[5] or WhoisXMLAPI[6]. However, the former provides about 555 requests per USD, and the latter provides about 833 requests per USD. Using such services would not be feasible if the system is to determine registration data for hundreds of thousands of domain names daily.

Domain Name Stat[7] offers a comprehensive and duly parsed historic WHOIS database for download. Such data are static, and even if updated regularly, they might systematically under-represent certain groups of examined names, such as hi-flux domains with limited lifetime used in botnets. Even so, a large number of queried domains could potentially be present in such database, limiting the need for online lookups. However, the database is not free and the pricing is not publicly available. The organisation was contacted twice to get more information, but no response was received.

4.3 Discovering Information Related to IPs

The maliciousness of a domain name is naturally strongly related to the maliciousness of the resources that the name represents. The most direct identifier of such resources is typically their IP address. Thus, it is beneficial to discover as much information about the IP addresses associated with a domain name as possible.

Different IPs can be found in DNS records: the A and AAAA records contain direct mappings of the name to an IP address, while other records, such as CNAME, MX or NS, contain names that can be further resolved to IPs. For each IP, much information can be discovered, such as the geographic location, the autonomous system number and prefix, reputation, or the liveliness of the address. Of course, the number of IP addresses of a domain name varies, and in the end, the information must be represented by a unified numerical feature vector. For this reason, the characteristics of all discovered IP addresses must be suitably aggregated. For example, the mean and standard deviation could characterise differences in round-trip times, while the Shannon entropy of a particular prefix could be used to capture diversity in the networks associated with the domain name.

Using reputation systems

IP reputation systems can also contribute to the classification of domain names. These systems evaluate and score IP addresses based on their history and behaviour, which may be beneficial in identifying potentially malicious internet resources. For example, domains

[5]Bulk Whois Api: https://bulk-whois-api.com/
[6]WhoisXMLAPI: https://whois.whoisxmlapi.com/
[7]Domain Name Stat: https://domainnamestat.com/

linked to IP addresses with a history of spamming or malicious activities may be classified as high-risk.

For the DomainRadar project, CESNET provided us unlimited access to their NERD reputation system[8]. It offers a comprehensive view of an IP's reputation by collecting data from various sources: reports from systems like honeypots, NetFlow analysers, and other detectors of malicious traffic, public blocklists, and several collaborative threat intelligence sharing systems [15].

Furthermore, the temporal aspects of IP reputation, as maintained by systems such as NERD, may provide insights into the evolving nature of threats associated with specific IPs and, by extension, domains. This could be beneficial in the future as the classification models will need to be updated.

Features: The reputation score could be directly used as a feature. The number of reports from various sources or the number of presences of the IPs in blocklists could also be used. Generally, the reputation system is expected to do a better job of devising a suitable scoring function; however, the NERD system specifically is still a work in progress in this regard. Features based on the date values associated with the records could also be calculated, similarly to what is proposed in the section on RDAP.

Using RDAP

The RDAP protocol can also be used to retrieve the registration information of autonomous systems and IP networks (or even nameservers and registrars) [89]. When querying for an IP address, the returned data may contain information (among others) on the network starting and ending address, the network name, and various entities, such as registrant, administrative, technical, or abuse contact.

In contrast to the situation with TLDs, RDAP data on IP networks is generally available, as all RIRs offer the RDAP service. The Number Resource Organization, a coordinating body for the world's RIRs, agreed on a profile that defines the requirements on data presented by RIRs in RDAP to increase consistency [92]. RIPE NCC [111] and ARIN [116] follow it; APNIC was reported to review its implementation to conform [81], but no current data are available; AFRINIC and LACNIC provide RDAP but do not state whether they conform to the profile [2, 71].

Features: The team experimented with features based on the lexical characteristics of the entities found for each IP, using the average length and entropy of names and email addresses. The shortest and longest network prefix lengths were also included. Other features could be derived from the network name or the associated date values, similar to the domain-based RDAP data discussed above. However, access to RDAP is subject to rate limits, and it should be carefully evaluated whether such features are helpful and cannot be extracted from static sources instead.

Discovering IP geographic location

The geographic location of IP addresses may offer other useful maliciousness indicators for domain names. For instance, certain regions may be more frequently associated with malicious behaviour, and domain names associated with IPs from such regions may be

[8]NERD: https://nerd.cesnet.cz/

classified as riskier. Moreover, the geographical spread of IP addresses linked to a single domain can indicate its operational scope. A domain associated with IPs from diverse geographical regions could be part of a global enterprise or service. In contrast, domains linked to IPs from a specific region could indicate a more localised operation.

Geographic data can be easily obtained from databases such as GeoIP2[9] by MaxMind. Their Country database offers information on the continent and country, while the City database contains more granular data on cities, including approximate coordinates. MaxMind claims that the data are 99.8% accurate at the country level. These products are commercial, but free versions branded as GeoLite2[10] are also available. The databases are updated weekly and can be downloaded or used through an API.

Features: To aggregate geolocation data across IP addresses, a unique identifier can be assigned to each possible combination of countries (*e.g.* by hashing the country codes and XORing them). Mean and standard deviation can be used for the latitudes and longitudes. It may also make sense to aggregate by determining a centre point by calculating the average of the highest and lowest latitude and longitude. The accuracy radii can be aggregated using the maximum value. The number of unique countries could can be used as a feature.

Determining the autonomous systems

An *Autonomous System* (AS) is a collection of IP routing prefixes under the control of one or more network operators, which collectively represent a single administrative entity with a clearly defined routing policy for the internet. Each AS is identified by a unique *Autonomous System Number* (ASN). A single organisation, such as an ISP, a large business, or an educational institution, often manages an AS. These entities control a specific set of IP addresses and announce their routing policy to other autonomous systems.

Examining the diversity and nature of autonomous systems associated with the IP addresses related to a domain name can also help infer its potential maliciousness. A domain linked to IP addresses spread across a wide range of autonomous systems, especially those with dubious reputations or known to host malicious activities, could indicate a higher risk profile. In contrast, domains associated with reputable and stable autonomous systems may be deemed safer.

IANA assigns ASNs in blocks to RIRs, who further assign specific ASNs or subblocks to other entities. The RIRs publish these assignments, though no standard format is used. Information on a specific AS can be retrieved using RDAP, but there is no unified way to determine the mapping between an IP and its AS. Fortunately, precompiled databases with such mappings are available, such as the MaxMind GeoLite2 ASN database[11].

Features: Autonomous system information can be used to evaluate the diversity of networks associated with a domain name. Features such as the number of distinct ASNs or average entropy of AS IP prefixes can be used. The GeoLite2 database also includes the name of the organisation that manages the AS, which could be used as a feature when hashed and XORed across the IPs, similarly to the country codes.

[9]GeoIP2: `https://www.maxmind.com/en/geoip-databases`

[10]GeoLite2: `https://dev.maxmind.com/geoip/geolite2-free-geolocation-data`

[11]GeoLite2 ASN: `https://dev.maxmind.com/geoip/docs/databases/asn`

Detecting the liveliness of the address

Another useful maliciousness indicator could be based on whether the IP addresses are actually reachable. This helps to assess short-lived services associated with malicious activities. Examining IP addresses using ICMP Echo (ping) involves sending a packet to the address and measuring the time it takes for a response (if any) to return. This *round-trip time* (RTT) is a valuable metric in assessing network performance and the responsiveness of the IP address. However, some networks may block ICMP messages, leading to false negatives.

The geographical and network location of the source of the ping influence RTTs. Therefore, probes should be sent from various locations to obtain a representative measure of an IP address's responsiveness. Such an approach would limit the possibility of skewing the RTT data by local network issues or geographical distance. However, this would require a large number of probes around the world, possibly introducing performance issues and increasing the cost of the data collection process.

In addition to ICMP Echo, other methods to determine if an IP address is active include attempting TCP connections on standard ports (like HTTP/HTTPS) or using more sophisticated network scanning tools. These methods can sometimes bypass restrictions on ICMP traffic and provide a more accurate picture of the IP address's status.

Utilising data from TLS handshakes

When a server offers HTTPS, the collector can establish a connection to the server, emulating what a user does in their web browser. This allows for gathering characteristics that identify the server and the service it provides. The TLS handshake offers information on allowed ciphersuites and protocol versions, which can serve as basic indicators of the server's configuration. More importantly, the server sends its certificate chain, which can be used to compute many potentially interesting features.

Typically, the HTTPS service would only be expected on the address found in the A/AAAA record or pointed to by a CNAME. It is not uncommon for a domain to have multiple addresses in these records, for example, to achieve load balancing. However, the service running on such an address is expected to behave the same, making it sufficient to only connect to one of the addresses. The collector could try a random other address if the first one does not respond, but it would probably not help to attempt connecting to all of them.

The collector should also indicate the target domain name using the SNI mechanism. This is important as many servers host multiple domains on the same IP address. Rational timeout settings should be used to avoid long delays in case the server does not respond.

Features: Several features based on the certificate chain were devised in the research. Key features include the length of the certificate chain, hash values of root and leaf CA names, validity periods of both leaf and root certificates, boolean values that signal invalid or expired certificates in the chain, or the counts of total and critical extensions. Another characteristic is the number of common names, subject alternative names, and unique second-level domains in the chain.

4.4 Limitations of the Prototype Data Collection Tool

When exploring the possibilities of phishing detection in the DomainRadar project (discussed in Section 2.3), I worked on collecting data for about 500,000 domain names using the *dr-collector* tool implemented by Horák in [57]. However, several functional weaknesses of this tool were discovered, rendering it impractical to use on large datasets or in the production DomainRadar pipeline. This motivated the work presented in this thesis.

The tool is implemented in Python and uses MongoDB[12], a document database, to store the domain name data. First, it loads domain lists from sources such as plain text or CSV to the database. Second, it performs the actual collection from external sources. The tool is designed modularly: the collection is handled using "resolvers", independent Python modules that accept a domain name and return a key-value structure (possibly nested) with the acquired data. The extraction of the feature vector is done using a separate tool.

The tool is affected by performance limitations that make it difficult to use with new extensive domain name lists. Although it allows for parallel processing of domain names, the implementation is flawed, leading to extreme memory usage and crashes. The entire process often gets stuck when all the worker threads encounter a domain name for which a resolver hangs. Also, it executes the independent collection steps (such as the DNS scan and the RDAP query) sequentially, hindering the performance.

Data collection issues

Multiple issues related to the implementation of the collection process were identified:

The DNS resolver works by sequentially resolving DNS records of various RRtypes. It only stores a basic, unstructured text representation of the records in the database, lacking information on TTLs. It only gathers IP addresses related to a name from the A records, while the AAAA and CNAME records are ignored. It always performs the resolution using pre-configured public resolvers, so the received data are not authoritative. It does not collect any DNSSEC-related data, nor does it determine the zone DN.

The RDAP/WHOIS resolver also always looks up the input domain name instead of the zones, even though subdomains are a considerable part of the input. RDAP will not return registration data for such queries – an RDAP domain object must represent a zone DN, that is, a point of delegation in the DNS [56, Sec. 5.3]. WHOIS behaviour in this regard differs across registrars. For this reason, many records end up not having registration data.

The resolver is prone to errors mainly due to the rate-limiting behaviour discussed previously, offering no means of preventing the issue. Because every domain name is processed independently, it cannot track connections to individual RDAP/WHOIS servers, and so it cannot determine when it should try to contact them again. Instead, it always tries to query for each DN, possibly further deepening the rate limit for its IP.

The TLS resolver uses the domain name as the target host, so another DNS lookup dependent on the OS configuration is always performed, instead of using data from the DNS collector. Two sockets/connections are created for no apparent reason.

[12]MongoDB: https://www.mongodb.com/

The socket is configured as blocking, which proved problematic when running the resolver on many domain names. With some of them, the attempt to establish a TLS connection makes the application hang for tens of seconds, even when timeouts are configured.

Furthermore, the system did not collect information on autonomous systems in which the related IP addresses reside. Although these could be easily determined even after the data collection is over, it would make sense to determine them during collection and store them within the dataset. Some properties of a domain name must be determined based on the zone DN, such as the SOA record or the registration information. The original version does not handle this at all. This wastes resources and adds to the rate limiting problem.

Erroneous states and re-collection

The tool works with two kinds of error during the collection from a data source:

The "resolution needs retry" error should represent a temporary failure, like when an RDAP request fails because of rate limiting. The collector may be able to resolve the missing fields when started again later.

The "resolution impossible" error should represent a permanent failure, like when the RDAP resolver cannot determine the correct RDAP server to find the object on. The resolver will probably never be able to collect the data in the (near) future.

Sometimes, it is ambiguous whether an error means the data cannot be resolved. This is apparent even from the implementation: for example, the ICMP resolver, which measures round-trip time using ICMP Echo, raises the retriable error when the remote host is unreachable, whereas the TLS resolver emits the resolution impossible error in a similar situation. This limits the possibilities of missing-field analysis and prevents the implementation of a more complex decision logic on how the lack of data shapes the classification outcome.

Furthermore, the tool does not track the individual retry attempts for missing data. This leads to a significant shortcoming observed in practice. When the process is started again, it always attempts to resolve all the missing fields, even when some domain names are apparently no longer reachable or if a particular error repeats at each run. This may seriously increase the execution time when collecting data for newly inserted domain names.

Chapter 5

DomainRadar and its Requirements on the Data Processing Subsystem

The initial goal of this thesis was to design and implement a data collection and feature extraction solution. It was always supposed to be a part of the DomainRadar system, but in the beginning, it was unclear what the system would look like. During team discussions parallel to the work on the thesis, we proposed the high-level architecture design of the system. It became apparent that the decisions taken in this thesis would significantly impact how this design would be realised. This chapter introduces the proposed architecture and the rationale behind some of the decisions. It also presents the requirements on DomainRadar as a whole and the implied requirements on the data processing subsystem.

Section 5.1 presents a data-flow description of how DomainRadar is split into several units and explains their role in the system. Section 5.2 discusses the expected throughput based on a sample of data captured in a real network. Finally, Section 5.3 summarises the requirements on the data processing subsystem based on the throughput expectations, the overall system's design and the experience from past experiments.

5.1 The Proposed Architecture

Overall, DomainRadar is a system whose input is a stream of domain names captured in a network coming from an external source. For each, the output is an in-depth classification report that shows the probability of the DN carrying phishing or malware or of being a DGA domain. DomainRadar's classification is based on the idea of using a large number of features derived from diverse related data, implying the need for a solid data collection unit. The ingress rate will reach hundreds of thousands of domain names daily (as discussed in Section 5.2), so it is not feasible to collect all the data for each domain name every time. Thus, it is necessary to include a process that omits previously classified and undoubtedly benign or malign domains from processing.

This hints at a possible separation of the system into four main units that, conceptually, form a data pipeline, as shown in Figure 5.1:

1. In the beginning, there are various sources of domain names to process. They may be imported from IDS/IPS systems such as Suricata, the network's DNS resolvers, or firewalls that parse DNS traffic, etc.
2. The *loader & pre-filter* perpetually fetches new data from the sources and decides which domains should be classified based on (persisted) previous experience with the domain, blocklists, allowlists, etc.
3. The *collector* gathers data from the external sources.
4. The *feature extractor* processes the raw collected data into a feature vector.
5. The *classifier unit* uses one or more pre-trained models to classify the DN.
6. The classification results are persisted in data storage.
7. An interface to access the results, be it a web application or an API, is provided.

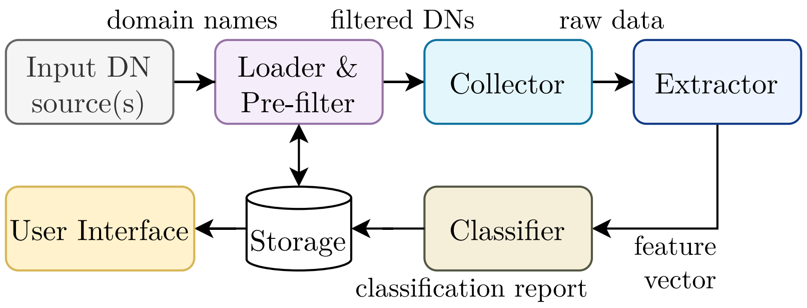


Figure 5.1: A trivial data-flow scheme of the DomainRadar pipeline, showing its main components – units.

Domain names ingestion

In the current design, most input domain names will be sourced from Elasticsearch, a distributed search and analytics engine [45], to which the actual network probes save reports that carry the DNs to classify. However, the loader must be designed with future extensibility in mind, so that it can be easily adapted to accept inputs from other sources. Currently, it is planned to support ingesting events from MISP, an open-source threat intelligence sharing platform [83].

In order to reduce the amount of data to process, the loader decides whether each received domain name should be further evaluated based on several factors: for example, if and when it was last seen, the inclusion in allowlists and blocklists, or fast and highly permissive lexical-only classification models that can detect "obvious" DGA-generated names. In addition to passing its output to the collector, the loader must also store per-DN metadata, such as the ingestion time, in data storage.

Some input domain names should be omitted from processing and storage. For example, to reduce noise in the interface, a user might define a wildcard blocklist entry that excludes randomly generated subdomains of a particular well-known domain. These domains should not be stored in the database as they would pointlessly consume space. However, if a blocklist is generated based on a MISP feed, it is helpful for the user to see that a domain was not classified due to its presence in a blocklist. Therefore, the loader can store the domain name and the reason for its omission (based on the configuration of the triggered filters).

The loader is designed as a modular standalone application. Figure 5.2 shows that it works with sets of sources (such as the ELKSource consuming from Elasticsearch or MISP source for the MISP platform), filters (such as the blocklists), and outputs (such as the persistent data storage). Where possible, it employs multi-threading or multi-processing to keep up with the input rate. The filters are independent, so they can run in parallel. Some sources can be partitioned and processed by multiple readers. Inputs are scattered across the instances running the filters, then gathered again to evaluate the filters' decisions and possibly pass them to the outputs.

Suppose testing shows that the loader cannot keep up with the input. This design makes it possible to migrate these independently running blocks to a distributed platform such as Apache Flink, as its computation can be represented as a DAG or even as a map-reduce operation. In order to make this future transition more straightforward, the operations should be stateless where possible and implemented with the use of I/O abstractions, preventing tight coupling.

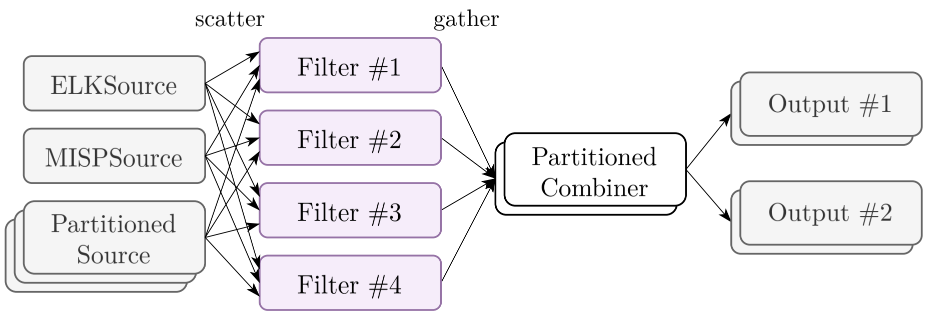


Figure 5.2: The loader & pre-filter unit. Each stage may be realised as an independent process (implemented, for example, through multi-threading or multi-processing). The use of partitioning may also help parallelise some stages.

Handling missing data

Quite often, collecting all the external data for a DN is impossible. There are three base approaches to this situation:

1. the DN is flagged as unclassifiable;
2. a partial feature vector is used for classification;
3. the DN and the already collected data are persisted, and the missing data are marked for a collection retry.

Especially when there are many external sources, the trivial approach (1) would lead to a significant dropping of inputs, rendering the system useless. Approach (2) requires classifiers that cope with missing data and may lead to a decrease in classification accuracy (and thus a higher false positive amount). Approach (3) may cause the system to be congested with retry entries. Apparently, a combination of all should be used: the system should be able to provide (appropriately flagged) lower-quality results based on partial data while also trying to obtain the missing data in the background. These attempts must be limited (*e.g.* by the number of retries or time), eventually marking the DN unclassifiable (or rather as “cannot be classified more accurately”).

Therefore, a supplementary unit that represents this logic, the *re-collection controller*, must be included as an additional source for the collector. The collector must store information on the processed names and the associated collection failures in data storage. The controller checks this information at certain intervals and triggers missing data collection. The collection process is eventually either completed or marked impossible for each external source (per DN).

Latencies

Most of the data processing is primarily bound by I/O: For each domain name, the system must store and retrieve metadata from the system’s database, and related data from external sources must be collected, which can take seconds of I/O waiting to complete. According to the needs of the project’s application guarantor, the tool is *not* required to process the domain names in real time – a delay of several hours or even days would be acceptable. Keeping low latencies inside the system’s components is thus not critical.

One of the considered options was to queue the domain names throughout the day and process them in a single large batch at night when the throughput is low. However, this assumes that the batch can finish in a reasonable time, which is not guaranteed, especially when some of the external sources may not be available. It may be especially unwise because of the rate-limiting issues of some data sources.

It is more natural to look at the system as a streaming pipeline that processes entries perpetually, as soon as possible, after they are ingested (with respect to the behaviour of the collector and the external sources). When missing data are to be re-collected, it is done simply by inserting a collection request at a certain point in the pipeline. In order to keep up with the ingress rate, the system needs to heavily parallelise the collection process to avoid getting congested with queued requests. It should be easily scaleable up or down based on the needs of the target deployment environment.

Other considerations

Several other critical requirements for the system architecture were identified. Firstly, all data with which the system works should be stored in a structured way, including the inputs, the decisions made by the filters, the collection results, and the classification results. The design of this storage should allow for easy querying and analysis. However, attention must be paid to the persistence time: when used in DomainRadar, it may not be necessary to store all the data indefinitely. The persistence behaviour should be configurable to a large extent. Similarly, all units should store operational logs in a unified and structured manner. The system should be able to log possible error states in all pipeline stages.

Secondly, users should be able to configure each part of the system from the UI. The configuration should be stored in a centralised place and passed to the units as needed. The user should also be able to trigger some actions from the interface, such as re-classifying a domain name. This may be achieved using a centralised communication bus (such as a message queue) that allows unified communication from the UI to the units.

The resulting scheme

Figure 5.3 shows a more detailed data-flow scheme of the DomainRadar pipeline, on which the team agreed. It reflects the requirements discussed above. It shows that most units in the system may be parallelised and suggests that distributed data storage could also be used if a single instance does not meet the throughput requirements.

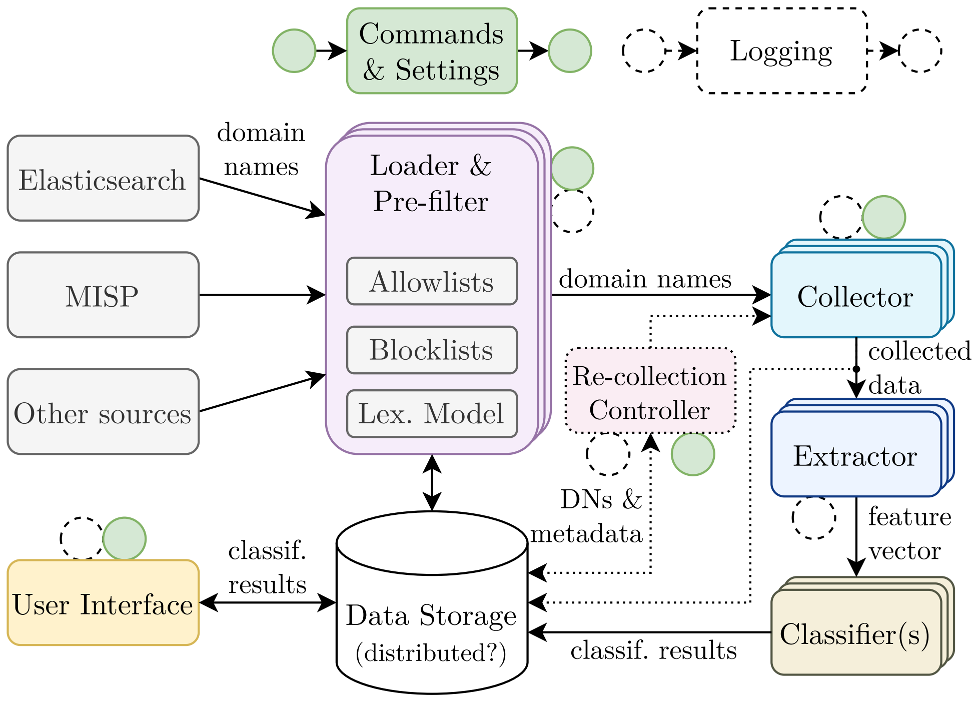


Figure 5.3: This conceptual look at DomainRadar's architecture shows the main units and the flow of data across them. The dotted parts represent the logic used when some external data are missing and must be re-collected later. The dashed circles represent logging. The green circles represent that a unit is connected to the configuration/command bus.

5.2 Expected Throughput Assessment

When used in the DomainRadar system, the loader unit will largely pre-filter the high-frequency stream of domain names used as the input for the collector unit. Using data from real-life traffic, we can make an upper-bound estimate for the throughput of the collectors and speculate on the possible effects of the pre-filter.

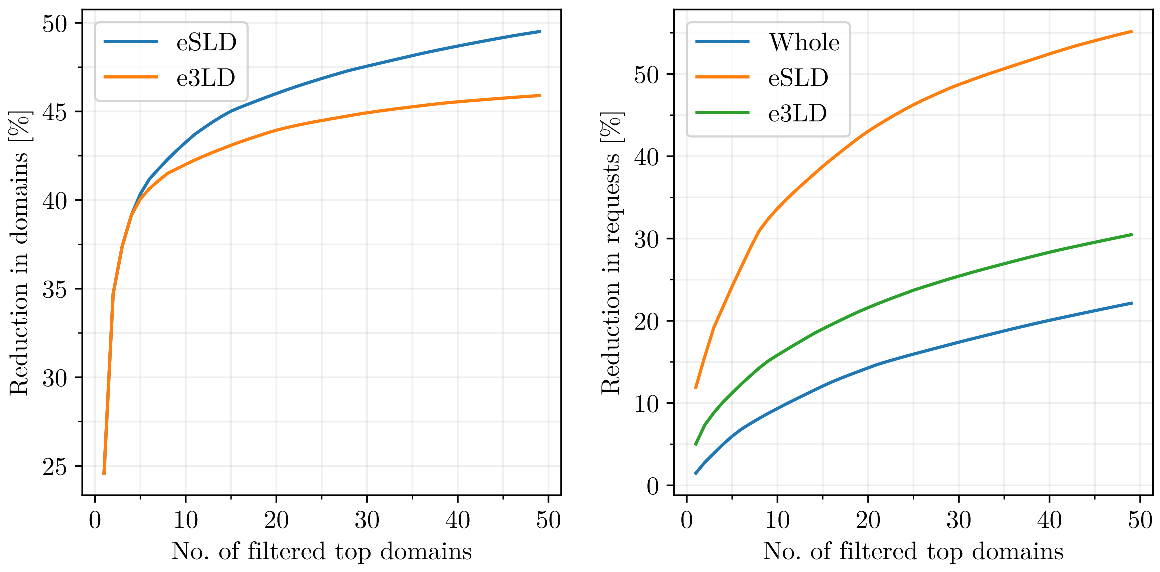
To make the following estimates, I used data provided to the DomainRadar team by CESNET, an association of 27 academic institutions in the Czech Republic that operates the national e-infrastructure for science, research, and education, including an extensive computer network [24, 25]. The nonpublic dataset comprised 56 lists of domain names resolved in the network, each encompassing the traffic from a single day. They were collected every day in February and March 2023 (three data files were corrupted and thus excluded).

The average total number of captured resolution requests per day was over 647 million. (This is mainly important for the loader unit, as it shows the raw amount of input domains it must process.) Among these, there were, on average, 1,840,563 unique domain names every day. This is the upper estimate of the input to the collector. Looking further, only

570,828 unique eSLDs and 912,631 unique e3LDs[1] were observed every day on average. Table A.1 in Appendix A shows the top 15 individual domain names, e3LDs and eSLDs in the dataset, sorted by the average number of resolutions per day, while Table A.2 shows the top e3LDs and eSLDs according to the number of unique subdomains.

The data show that a considerable portion of requests target domains of large service providers such as Microsoft, Apple, Facebook, or Google. Moreover, both the top DNs and the top e3LDs contain many domains that are likely not accessed directly by users but instead in the background processes of applications. This is even more apparent in the e3LDs and eSLDs with the most unique subdomains: almost a quarter of the average total daily unique domains are subdomains of `safeframe.googlesyndication.com`, and another 10% are subdomains of `measure.office.com`. Large vendors use these generated domains for various purposes, such as tracking, analytics, or advertising. Such domains can be safely omitted from further processing (filtered out by the pre-filter) without evaluation. This shows potential for helping considerably reduce the input to the collector.

Figure 5.4a shows the approximate level of reduction in the number of domain names that the collector would process if a certain amount of top e3LDs/eSLDs, sorted by the number of subdomains, were filtered out. Even omitting the top 10 e3LDs could reduce the number of unique domain names by more than 42%. For comparison, Figure 5.4b indicates the effects of filtering out the top N domains, e3LDs and eSLDs, sorted by the number of resolution requests.



(a) Reduction in total observed domain names.
(b) Reduction in total resolution requests.

Figure 5.4: Approximates of reduction in observed domain names and total resolution requests when the top N domains/e3LDs/eSLDs are filtered out.

However, it still needs to be considered that the system will not re-evaluate the previously seen domain names every day. The loader unit will store the last time a domain name was classified and will only pass it to the collector if a certain amount of time has passed. Figure 5.5 shows how many new, unseen domain names would be added in the first 50 days in the dataset. Additionally, to show the effects of common DN filtering, the *Unfiltered* line shows this development in the original dataset, while the *Filtered* line shows

[1]When discussing e3LDs here, note that names with only two effective levels were also included.

the development if the top 15 e3LDs (pre-computed over the entire dataset) were always filtered out. The green line shows the percentage of domains filtered on that day calculated as $100(1 - \frac{\text{filtered}}{\text{unfiltered}})$.

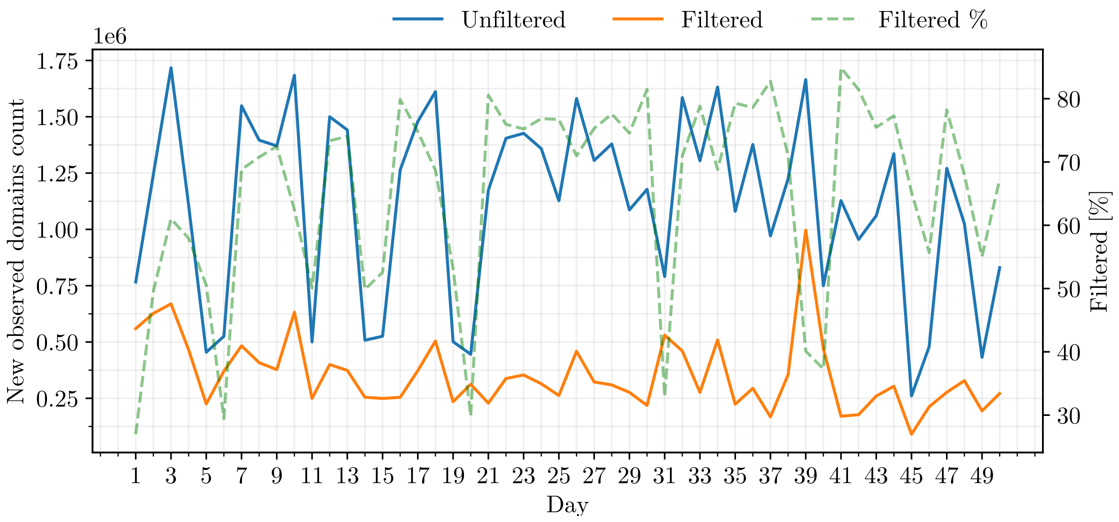


Figure 5.5: This plot shows how many not-yet-seen domains are added in each dataset day. The *Filtered* line shows the development if subdomains of the top 15 e3LDs (by subdomain count, see Table A.2) were always filtered out. The left y axis shows millions of DNs.

Two key observations can be made from this plot. First, it again shows how filtering the top e3LD domains can significantly impact on the number of domain names passed to the collector – the maximum filtration ratio was 85%. However, it also went as low as 27% on some days. In practice, the filtering process should be dynamic and update its top e3LD list over time, as new versions of commonly used benign software may use newly generated domain names. At the same time, it should not learn to filter out malicious software that behaves similarly.

Next, we can focus on the trend in total newly observed domain names (which are passed to the collector every day). Although the chart shows a certain decreasing tendency, it is not as substantial as one might expect, even after filtration. This suggests that the collection system should still be able to process around 300,000 domain names daily and withstand peaks as high as a million domains in a day.

5.3 Requirements on the Collector Unit

While the collector is supposed to be used as a part of DomainRadar, it should be designed with independent usage scenarios, *e.g.* the standalone collection of new datasets, in mind. Eventually, the addition of new data sources may be required, so the unit should be extensible. It should never idle when there are requests that can be processed, maximising the usage of resources dedicated to the system.

External sources

The collector seeks to gather data from sources described in Chapter 4 for each input domain name, keeping up with the expected throughput. It will:

- Determine the corresponding zone domain name, the zone's SOA and a list of its authoritative nameservers, including their IP addresses.
- Fetch registration information from RDAP, using WHOIS as a backup. The most important attributes are the registrar name, the registration date, the expiration date, and the last updated date. However, the entire RDAP or WHOIS response should be stored for further research and analysis.
- Query the authoritative nameservers for A, AAAA, CNAME, MX, NS, and TXT records on the input domain name. Adding support for other record types should be easy if later found useful. TTL values should also be stored.
- Try to establish a TLS connection to one of the addresses sourced from A, AAAA, or CNAME records and store handshake and certificate data.
- Extract related IP addresses from the records. The selection of record types to use should be configurable.
- For each IP address, fetch/determine:
  - the autonomous system information – the AS organisation, number, and network,
  - the RDAP-provided information – primarily the associated entities' data, but the whole RDAP response should be stored,
  - the geolocation data – *e.g.* the country, city, and coordinates,
  - the reputation score from the NERD system,
  - the ICMP Echo (ping) round-trip time (RTT).

In the case of the RTT discovery, it should be possible to use multiple probes in different geographical locations to limit the bias caused by the network topology. The RDAP-based collection processes must include mechanisms to prevent triggering rate limiting on the remote servers.

Note that there are two general categories of sources. *DN-based collectors*, such as the zone resolver or the registration information collector, only require the target domain name as input. *IP-based collectors*, such as those determining the autonomous system or reputation, instead focus on gathering data for a specific IP address. The TLS collector is a special case, as it connects to an IP address but also requires the domain name, as many servers use the Server Name Indication feature to facilitate connection to virtual hosts [43, Sec. 3].

Clearly, some of these processes are independent and should run in parallel. For instance, the DNS collector does not need to wait for the RDAP or TLS collector to finish. However, the DNS collector can only run after the zone has been determined, as it uses the domain's primary nameserver address. Moreover, the processes should be *independently scalable* in order to provide efficient resource usage. For example, the only dependency of the domain-name-targeted RDAP data collection process is the zone domain name. This process could thus be implemented together with the zone resolver. However, the zone-resolving process is likely to be faster, as it only performs a handful of UDP-based DNS queries. At the same time, the domain name RDAP data collector opens HTTPS connections and parses JSON responses. If this was a single process and it was to be scaled, there would be many unnecessary unsaturated zone resolving units.

Non-existent domain names

If a domain name does not exist (it cannot be found in the DNS), there is virtually nothing to collect about it. If the collector unit determines that a name does not exist, it should:

1. provide this information on the output; and
2. discard the domain name from further processing as soon as possible.

Passing this domain name with "empty" data to the feature extractor should be possible, as a lexical-only classification model could still provide some information about the domain name.

Missing data and error handling

Handling errors that lead to unavailable data from one or more sources is a problem at the boundary between the data-collecting unit and the rest of the DomainRadar system, represented by the re-collection controller (which is not considered a part of this work). The collector should not handle the re-collection process on its own. Instead, in order to integrate with DomainRadar or to be used in other ways, the collector only needs to:

1. handle errors gracefully – not crash;
2. provide enough information on the output so that the other units can make informed decisions; and
3. process each domain independently so that a failure in one domain cannot affect the processing of others.

As such, the collector output should include both the successfully collected data and a representation of the errors that occurred during the process. There should be a well-defined list of error codes that the collector can emit so that the re-collection unit can decide how to proceed with each individual name. The list should contain data errors (caused by the remote service's behaviour) and system errors, such as the inability to establish a connection to the remote service or other exceptions raised inside the collector. This way, the re-collection unit can implement various kinds of retry logic according to the needs of the individual deployments or use cases.

Additionally, the team is currently researching the possibility of using incomplete data for the classification. In the future, it should be possible to process incomplete datasets by the feature extractor and the classifiers. The feature vector should then include information on missing data so the classifier can make an informed decision based on the available data.

Inputs and outputs

The collector will accept a stream of domain names, allowing the use of various sources of domain names. In DomainRadar, the loader and the re-collection controller will provide input to the collector. In the standalone collection use case, the input domains may be loaded from a bounded source, such as a file. Each input entry must contain a string with the domain name for which to collect data. When performing re-collection, the input entry should also specify which collectors should be triggered, possibly together with other options. Some re-collection requests may be targeted at a set of related IP addresses instead. The outputs will be persisted appropriately and, if used in DomainRadar, passed to the feature extractor.

Configuration

The system must provide a centralised way of configuring the behaviour of the individual collection processes at runtime. The configuration must be stored and automatically applied in the event that the system or its part is restarted. The system must validate the configuration requests before applying them.

Generally, the system is expected to provide configurable time boundaries for external requests (timeouts) to limit the total time required to process a domain name. Rate-limiting settings must be provided, at least for the processes targeting RDAP. No other precise requirements are given on the configurable properties as they will be determined based on the implementation.

5.4 Requirements on the Feature Extraction Unit

The feature extraction unit will essentially implement a transformation function that accepts the data collected in the collector unit and outputs a feature vector in the form required by the classifier unit. Although the feature extraction process is not expected to be computationally intensive, it should be implemented with parallelisation in mind, as the input data for each domain name can be processed independently.

In order to provide the necessary reliability, the extractor must properly handle the cases of missing data from one or more collectors. It must never omit a feature from the output; instead, it must use a default value. If required by the classifiers, information on missing data can be included as another feature. Corrupted or otherwise erroneous data should not affect the computation of feature vectors for other inputs.

Notably, the extractor must be functionally equivalent to the prototype feature extraction code currently used in the DomainRadar research to be usable with the existing classification models. At least a partial re-usability of the existing code would be beneficial, mainly to avoid introducing errors when reimplementing the feature extraction logic and to make it easier for other team members to understand the new codebase.

Chapter 6

Design of the Data Processing Pipeline

This chapter introduces the design of the data processing pipeline that meets the requirements set previously, explaining the rationale behind the important choices. Section 6.1 presents a concept of the pipeline as a whole and discusses certain simplifications that lead to an appropriate level of parallelisation. Section 6.2 summarises the resulting design. Section 6.3 lays the foundation for the specifications and describes the shared configuration exchange mechanism. Section 6.4 provides an overview of the collector components, introducing the specifications and interfaces of the components defined in detail in Appendix B. Section 6.5 explains the operation of the data merging component. Section 6.6 specifies the interface and process of the feature extraction process and touches on the integration with the classifier unit. Section 6.7 drafts a simple algorithm that could be used as the re-collection controller logic.

The data processing pipeline is designed as a series of logically independent "input–transform–output" stages (components) with well-defined behaviour, including request/response data models and error states. The stages must be implemented on top of a platform that will manage the data flow, scaling out, and distribution of work between the running instances of each stage, as well as interfacing with external storage (in this case, Apache Kafka will be used). This thesis focuses on two sections of the system: the *collection* stages (also called "collectors"), retrieving the required data, and *extraction* stages, computing the target features from the data.

Each collector corresponds to one of the external sources described in Chapter 4 (DNS, TLS, RDAP, etc.). The extractor stages can be considered transformation functions that accept a subset of the collected data and output a part of the feature vector. The transformations include data normalisation, feature encoding, or statistical analysis. They are based on the existing design from the DomainRadar research and mainly compute the features described in the same chapter.

6.1 Parallelism Granularity and Data Gathering

The scheme in Figure 6.1 shows the overall flow of data in the system (induced by a single input) highlighting the interactions between various stages and data synchronisation points. The collection and extraction processes are represented using a set of individual stages that exchange data by accepting requests and producing responses, as well as requests for other stages. The scheme works with a generalised source of domain names, making it pluggable into various environments. Note that the scheme only shows that classification results are persisted. A solution for the persistence of the collected data will be presented later.

This specific decomposition of the data collection processes (shown in green and blue in the scheme) was chosen by representing each remote data source with a separate collector. This way, the design addresses the independence of data sources, which is also a step towards scalability independence. However, this property must be ensured by the underlying platform that will manage work distribution among them. The scheme does not show this, as it focuses on the overall data flow for a single input.

Observe that the scheme shows several points where synchronisation is needed between the data flowing from the parallel processes – *i.e.* where the data from independent sources merge. The impact of these operations is discussed below.

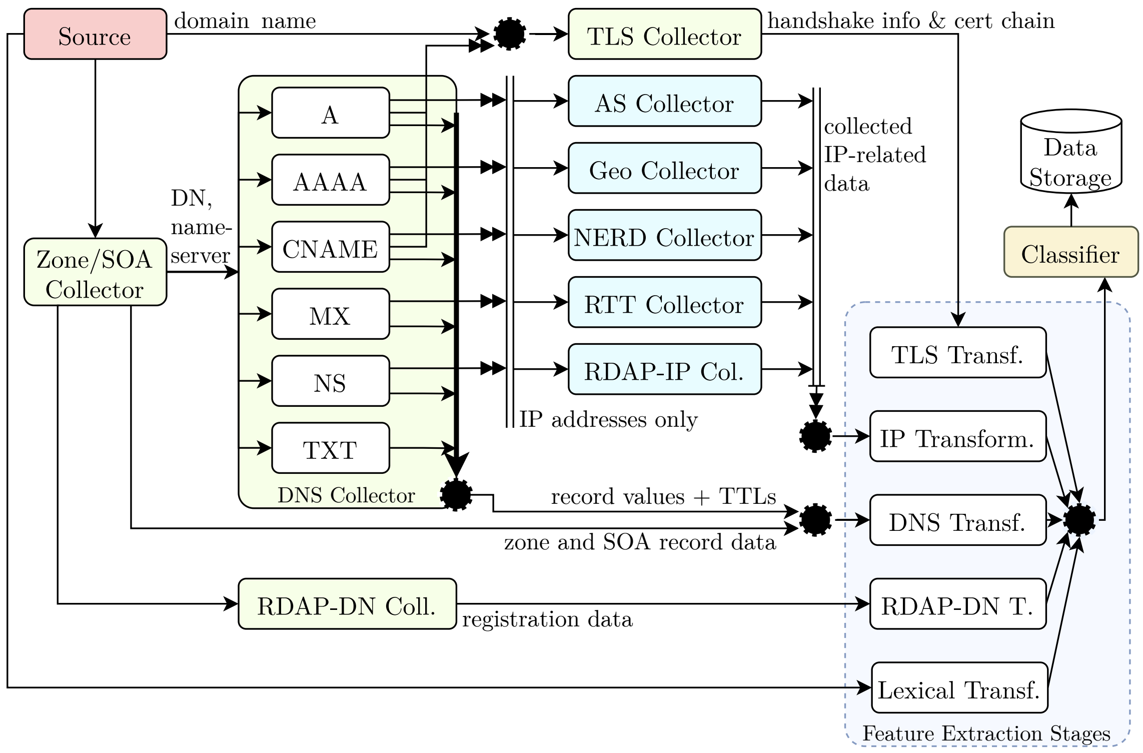


Figure 6.1: A data-flow scheme of the data processing pipeline. Green blocks represent collectors that only fetch data based on a domain name, and blue blocks represent collectors that fetch data based on an IP address. Double arrows signify that a stage may produce multiple output values (IP addresses) for a single input. The black circles represent a "Gather" operation over data from multiple sources. The grey dashed block includes the independent feature extraction stages.

In the feature extraction part, the possible granularity depends on the relations between the input data and the output features. The scheme identifies five possibly independent transformation functions:

- the TLS transformation only uses data from the TLS collector;
- the IP transformation aggregates data from all the IP collectors, necessitating a Gather operation over all the results for all the IP addresses;

- the DNS transformation computes DNS-based features, and so it requires a complete set of DNS records for a domain name, necessitating a Gather operation on the individual DNS record type results and the zone/SOA results;
- the RDAP-DN transformation only uses data from the RDAP collector;
- the lexical transformation is not dependent on any collector as it computes features only from the textual representation of the domain name.

The Gather operation

While it is possible to split the entire process into a large number of independent stages, it is not always beneficial to do so because the system works with a high number of data dependencies – data coming from independently running stages must be merged at some point in order to provide a unified input to the classifiers. For this reason, the data flow in the diagram in Figure 6.1 includes several "Gather" operations (shown as black dashed circles) that join the data from multiple sources based on a shared key – the domain name. Naturally, these operations also work as synchronisation points in the parallel system.

Here, we shall model the operation as an input/output process with an internal state represented by a map of keys to sets of values, in which the following properties hold:

1. Each value in the sets and each output value are labelled with an identifier of its producer.
2. The operation has a priori knowledge of all the expected labels per key.
3. The number of entries in the map is unlimited.
4. When a value with a previously unseen key is received, the key is inserted in the map with an empty value.
5. When a value is received, it is placed in the set associated with its key (possibly just created as per Property 4).
6. When all the expected values for a key are present, the map entry is produced on the output.

Obviously, an implementation cannot provide an infinite entry storage. The model can be extended with certain constraints that limit the lifetime of an entity, such as the maximum time since the first introduction of a key or since the last update. This should not affect the correctness of the system, as it is expected both that:

- all the collecting stages will respond in a reasonably short time frame (perhaps with an error); and
- all the data available at a certain point in time will be collected in a reasonably short time frame.

Suppose one of the input stages produces multiple results for the same key while it is present in the Gather operation's internal state (in the given time frame). In that case, the sets must be updated appropriately. Thus, we add two additional properties:

7. When a newer value is being inserted in the set, it replaces the older value with the same label, if present.

8. A map entry is expelled after some time. If an entry is being expelled before it has been produced, empty values for the missing labels are inserted in the set, and the entry is first produced on the output.

This way, it is guaranteed that an input domain name will eventually pass through the entire pipeline. If a later attempt at re-collecting some data is to be made, all the values for the key will either still be present in the Gather operations, or reintroduced into the system from a persistent storage.

Towards a simpler model

The Gather operation introduces synchronisation overhead, primarily in terms of latency and resource usage, which is why each occurrence should be evaluated. The model shown in Figure 6.1 uses several Gather operations that can be optimised out in practice, simplifying the model.

First, note that the input domain name must piggyback on all the messages flowing through the pipeline because it is used as the key in the Gather operations. This means that no gathering actually exists in front of the TLS collector: the necessary domain name information will be passed together with the input address from the DNS collector.

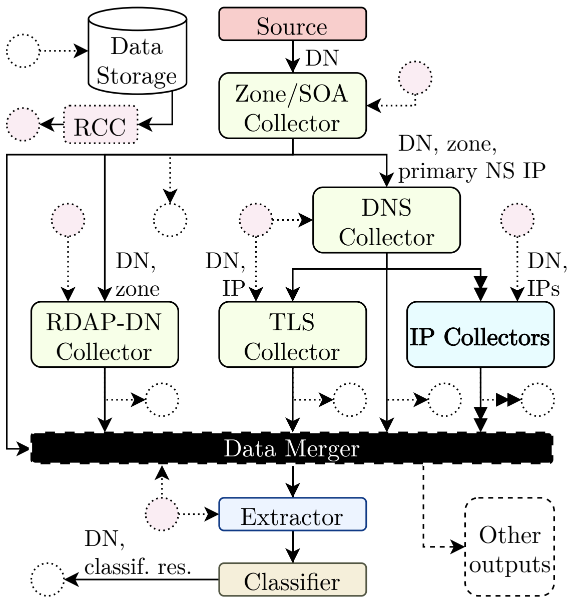
Another usage of the operation is in the DNS collector. If the records are fetched in parallel, some synchronisation or gathering is undoubtedly needed, as the subsequent transformations work over the complete set of DNS data. The DNS queries are likely the fastest remote fetches in the pipeline, so performing them in a single process should be feasible, rather than considering the fetch for each record type as a separate pipeline stage. This way, the Gather operation will be implicitly present inside the DNS collector itself, converting it into an implementation detail. For example, the per-query parallelism may be achieved using concurrency-based programming techniques based on asynchronous I/O.

The collection process for a single domain name is generally expected to take significantly more time than feature extraction. Using a highly granular model for the extractor would necessitate several Gather operations, and the overhead of managing the parallel computation may outweigh the benefits. Even sequential execution of all the transformations (per a single input entry) may be sufficient and even preferred because of the ease of implementation and maintenance. In addition, it makes it easier to ensure the data's consistency and completeness and simplifies the feature extraction's extensibility. Introducing new features will require no modifications to the pipeline, even if they require data from multiple sources. If necessary, the unified feature extraction process can be "explicitly" parallelised per domain name (*e.g.* by using multiple threads inside of a single instance) or by scaling out.

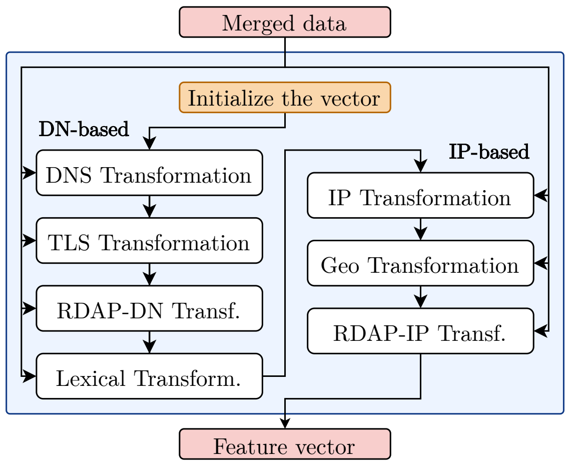
6.2 The Final Pipeline Model

Figure 6.2 shows a scheme of the final design of the pipeline, adjusted based on the findings from the previous section. Every message in the pipeline carries some representation of the input domain name, the source component that produced it, and the time it was produced. All collector results also include an indicator of error, if any occurred.

The collection part of the pipeline starts with the zone/SOA collector that determines the DNS point of delegation and the authoritative nameservers. It provides requests for the DNS collector that performs a DNS scan and for the RDAP-DN collector that retrieves



(a) The adjusted pipeline. The pink dotted circles represent points where the re-collection controller (RCC) may inject messages. The white dotted circles represent persisting data in storage.



(b) The extraction process is composed of transformation functions that sequentially populate and modify the feature vector, with access to the unified input data object.

Figure 6.2: The final scheme of the data processing pipeline that simplifies the data-gathering operations into a single component.

registration data. When the DNS collector resolves IP addresses related to the domain name, it selects an address passed to the TLS collector. It also passes all the addresses to an IP collector input bus. Various IP collectors (omitted in the figure for clarity) listen to the bus, process the addresses independently, and publish their results to an output bus. The collectors' result data are processed by a single "data merger" stage that implements the Gather operation over all the data collected for a domain name.

All collection results are also persisted in data storage. The re-collection controller periodically checks the erroneous results and schedules re-collection attempts by injecting requests in virtually any place in the pipeline, while considering backoff intervals and retry limits to manage repeated failures. For example, if the TLS collector fails to establish a connection, the controller will eventually execute a re-collection attempt by sending a request to the TLS collector. Results from the other collectors could have already been removed from the data merger's state, so the controller must also inject the already collected data back into the merger. Suppose domain names were to be re-classified based only on previously collected data (*e.g.* when the classification models are updated, especially during development and debugging). In that case, the re-collection controller may inject the unified data object directly into the extractor. Note that the scheme also contains a dashed "other outputs" block. This output channel may be used to store the merged objects when collecting new datasets, for debugging, or for future extensions of the pipeline.

The feature extraction process is consolidated into a single pipeline stage to streamline data processing and simplify the system architecture by minimising inter-stage communication. Its input is the merged data object. Internally (Figure 6.2b), the process prepares a feature vector structure and then passes it through a series of transformation functions that have access to the original data object and use its data to populate the feature vector. Finally, it is passed to the classifier stage. The figure shows that the transformations have access to the original data object – the implementation will pass it from one transformation to another to keep the process sequential.

6.3 Common Definitions for the Functional Specifications

This section defines the type system and base models used in the specifications of the pipeline stages and their input/output models given throughout this work. It also describes a unified configuration mechanism for all the pipeline stages.

Type system

The model definitions use a simple type system based on Python. The base data types are described in Table 6.1.

`object`	any non-null value	`int`	a signed integer
`float`	a floating point number	`bool`	a truth value
`str`	a string of characters	`bytes`	a string of bytes
`timestamp`	a UTC date and time	`ip`	an IPv4 or IPv6 address
`list[T]`	an ordered list of values of type `T`		
`set[T]`	an unordered set of unique values of type `T`		
`dict[K, V]`	an unordered mapping from keys of type `K` to values of type `V`		
`json`	a JSON-serialisable value		
`feature`	a feature vector value		
`fvector`	a feature vector		
`None`	an empty (null) type and value		

Table 6.1: Base types used in the data models.

The | operator denotes a union of types. If a type of a field is not a union with `None`, its value must not be null (`None`). It is up to the implementation how the types are represented, what their exact bit lengths are, etc.

The special `json` type represents a complex value that could be serialised as a JSON [44] (without imposing any restrictions on the actual representation):

```
json := str | int | float | bool | None | list[json] | dict[str, json]
```

The `feature` and `fvector` types used in the feature extractor are defined as:

```
feature := int | float | bool | None
fvector := dict[str, feature]
```

Data models

The data models specified in this chapter and Appendix Section B.11 are described using a syntax similar to Python dataclasses:

```
class Model:
    fieldA: Type = default
class InheritingModel(Model):  # also has fieldA
    fieldB: TypeA | TypeB
```

A model may inherit all properties from a parent model. The default value may be omitted; if the value can be `None`, the default value is `None`, unless specified otherwise. A value must always contain all fields specified in its type and no others.

All messages consumed and produced by pipeline stages are key-value pairs. Both the key and the value are of a specific type. Each stage has a main output channel and may define side output channels. Most stages will subscribe to an output channel of another stage.

Collector base result

The base model of a result message for a collection stage is `Result` (shown in Listing 6.1). Each collector uses a derived model with its specific fields that carry the actual result data. A result message contains an integer status code that represents the result of the collection. The value of 0 means success. All status codes are defined in Appendix Section B.13. All collectors may return any of the codes defined in the "General status codes" section of the table therein; error codes specific to each collector are stated below. The `error` field may contain a human-readable error message if (and only if) the operation was unsuccessful. The `lastAttempt` field contains a timestamp of when the operation was finished.

```
class Result:
    statusCode: int
    error: str | None
    lastAttempt: timestamp
```

Listing 6.1: The base model of a result message.

Configuration and its runtime exchange

This design specification does not include a detailed description of the configuration options for the stages, as it is considered an implementation detail. The configuration is expected to include options such as the maximum number of retries, timeouts, or the number of parallel requests. It is expected that the stages will have a static part of the configuration, which will be considered a part of the deployment, and a dynamic part of the configuration that controls the behaviour of the stage and can be changed by users at runtime. The implementation must ensure that the dynamic configuration is stored in persistent storage and loaded by the stage at startup.

A configuration object is a `dict[str, json]` value. A configuration property can be addressed using a natural dot-separated path notation – for example, the property `foo.bar` in a configuration object `{"foo": {"bar": 42}}` would have the value 42.

The components provide a shared runtime configuration exchange mechanism. All stages offer a side input channel that accepts configuration messages and a side output channel for configuration change result notifications. The key of a configuration message is a `str` that identifies the target stage; the value is the complete configuration object.

The stage validates the received configuration object and, if valid, replaces the current configuration with the received one.

After a stage processes a configuration message, it must publish a configuration change result message. Its key is the same as the configuration request message key. The value is a `ConfigurationChangeResult` defined in Listing B.20:

- `currentConfig` always contains a complete snapshot of the stage's configuration.
- `success` is true if (and only if) the configuration was successfully applied.
- `errors` may contain a list of `ConfigurationValidationError` values that describe the individual errors (or warnings) per property. The `errorCode` contains one of the values from Table B.9.
- `message` may contain an arbitrary human-readable message that describes the result.

There might be multiple errors for a single property. A validation error may be "soft" – that is, it did not prevent the configuration from being applied but may cause issues. In this case, the `ConfigurationValidationError` instance is interpreted as a warning.

The last result message from each stage is stored. When a stage starts, it must load its runtime configuration by fetching this last result message and applying the configuration from the snapshot in `currentConfig`. If the message is not found, the stage must apply its predefined defaults and publish a change result message as if the configuration has just been changed. If the snapshot does not contain a valid configuration (*e.g.* the format of the configuration has changed), the stage may either apply the configuration partially or use the defaults.

6.4 The Collectors

This section provides a brief overview of the collectors' operation. The detailed specifications of the collectors are provided in Appendix B.

The key of the request messages accepted by the zone, DNS, TLS, and RDAP-DN collectors is always a `str` value that contains a domain name. The key serves as a correlation identifier for gathering the results for a single input domain name. The value of the request is collector-specific. A response sent to the main output of a collector must have the same key. The target for a collector's lookup may be selected using the key, the value, or both.

The key of messages accepted by the IP-based RDAP-IP, NERD, RTT and GEO-ASN collectors is always an `IPToProcess` value, a pair of a domain name and an IP address (see Listing B.1). Both IPv4 and IPv6 addresses are supported.

Zone collector

The zone collector accepts the domain name and determines the domain name of the DNS zone that contains the input name by finding an SOA record using Algorithm 4.1. A pre-configured recursive DNS resolver is used for all DNS queries. If the SOA is found, additional queries are made to check the presence of a DNSKEY and to obtain:

- the A and AAAA records for the primary nameserver,
- the NS records for the zone,

- the A and AAAA records for all the secondary nameservers from the NS records.

The zone collector typically subscribes to an external source of input domain names to process. See Appendix Section B.1 for the specification.

DNS collector

The DNS collector queries the primary nameservers of the input domain name for records of the requested or pre-configured types and collects their values. It also finds the target IP addresses using a pre-configured recursive resolver for record types that carry a hostname (CNAME, MX, NS). The DNS collector typically subscribes to the *DNS requests* output of the zone collector. See Appendix Section B.2 for the specification.

TLS collector

The TLS collector opens a TCP connection to an input IP address, port 443, attempts to perform a TLS handshake, and, if successful, outputs a message that contains the used protocol, the ciphersuite identifiers, and a list of DER-encoded certificates presented by the server. The input domain name is used as the SNI value. The TLS collector typically subscribes to the *TLS requests* output of the DNS collector. See Appendix Section B.3 for the specification.

RDAP-DN collector

The RDAP-DN collector uses the Registration Data Access Protocol to look up domain registration data. Mappings of TLDs to RDAP endpoints are obtained from IANA[1]. The legacy WHOIS service is used as a fallback in the event that the TLD does not provide RDAP access or when an error occurs. The RDAP-DN collector typically subscribes to the *RDAP-DN requests* output of the zone collector. See Appendix Section B.4 for the specification. Note that the collector implements a local rate limiting mechanism.

RDAP is expected to provide data only for registered zones. The request body should contain the zone domain name for the RDAP query (*e.g.* determined by the zone collector). If the target is not provided, the collector will use the input domain name (key) as the target. If the query fails, the collector tries the possible registered name – obtained by keeping only the public suffix and the first label that comes after it – if it differs from the previous target. Only after the failure of this second query (if it was made) is the WHOIS query attempted. Only a single WHOIS query is performed, targeted at the zone DN, if provided, or the input (key) DN otherwise.

IP-based Collectors

All the IP-based collectors are expected to consume messages from an input bus and produce messages to an output bus. For this reason, each collector type is assigned a unique identifier used to signalise the target/source collectors (see Appendix C). The `IPRequest` request message (see Listing B.1) may contain a list of IP collectors that should process the input.

[1]The IANA Bootstrap Service Registry for Domain Name Space: https://www.iana.org/assignments/rdap-dns/rdap-dns.xhtml

RDAP-IP collector

The RDAP-IP collector uses the RDAP protocol to look up IP registration data. It fetches the mappings of IP network prefixes to the RDAP endpoints from IANA[2]. The collector does not provide a WHOIS backup. See Appendix Section B.6 for the specification.

NERD collector

The NERD collector retrieves the reputation score for the input IP address from CESNET's NERD reputation system. See Appendix Section B.7 for the specification.

GEO-ASN collector

The GEO-ASN collector looks up information on the geographical location and autonomous system of the input IP address in MaxMind's GeoIP databases. See Appendix Section B.8 for the specification.

RTT collector

The RTT collector performs a standard ping: it sends several ICMP Echo messages to the input IP address, waits for the ICMP Echo Reply answers, and outputs basic statistics. See Appendix Section B.9 for the specification.

6.5 Merging the Collected Data

The final step in the data collection phase of the pipeline is the data merger. This process performs the Gather operation, combining data from various sources into a single coherent data object for use by the feature extractor. The input messages are the outputs from the collectors, and the output message is a value that encompasses all the gathered data for a specific domain name.

The merger builds a *most useful state store* for each input channel (*i.e.* collector output). The store is a key-value mapping where keys are the domain names and values are containers populated with the *most useful* results for the DN from the respective collector. The "B is more useful than A" relation $\succ$ compares two candidate results A, B based on the timestamp and on whether they were successful. It is defined as:

$$B \succ A \iff (\neg A_{OK} \land B_{OK}) \lor (\neg A_{OK} \land B_T > A_T) \lor (B_{OK} \land B_T > A_T).$$

where A_T and B_T are the respective values of the `lastAttempt` field, A_{OK} is true iff the status code of A is 0, and B_{OK} is true iff the status code of B is 0. Note that this relation has been created by the minification of the rules specified in Table 6.2.

Whenever a message is consumed from one of the inputs and the value is more useful, the corresponding container is updated. The store is regularly cleaned to remove entries that have not been updated for a long time. The behaviour is the same for IP-based collectors, but the IP address *and the collector identifier* are parts of the key. The identifier is included to differentiate between the results from different collectors sent to the shared output bus.

[2]The IANA Bootstrap Service Registry for IPv4 Address Space: `https://www.iana.org/assignments/rdap-ipv4/rdap-ipv4.xhtml`. Similarly for IPv6: `https://www.iana.org/assignments/rdap-ipv6/rdap-ipv6.xhtml`.

A_{OK}	B_{OK}	$B_T > A_T$	$B \succ A$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Table 6.2: A truth table capturing the rules for comparing the usefulness of two results from a collector.

The scheme in Figure 6.3 demonstrates how the data merger operates using an object-relational model. The most useful state stores are represented by "tables", where the domain name is the key and the message data object is stored in the `result` attribute. For IP-based collectors, the key comprises the domain name, the IP address, and the collector identifier. The collectors' output channels are connected to the state stores, which are updated with each message. When a state store is updated (*i.e.* an entry is modified with a more useful result), the entire operation is re-executed for the domain name.

The first grouping operation takes the IP collector results, groups them by pairs of domain name and IP, and aggregates the resulting "rows", pairs of collector ID and result, into a complex key-value map object `collectedData` using the `COLLECT_TO_MAP` operation. The second grouping creates groups of results from all collectors for a single domain name. Here, the operation aggregates the resulting pairs of IP address and the `collectedData` object, into a single complex map of maps.

A result should only be produced after all the collectors have sent their results for a given domain name. The inner joins between the zone results, DNS results, and RDAP-DN results ensure that an entry will not be included in the output if any of these collectors fail to provide the data. However, it is valid to produce a result with missing TLS or IP data, as they only exist if the DNS collector succeeds. Thus, (left) outer joins are used to add TLS and IP data to the result. The filter operation explicitly discards results that have DNS data but not the expected IP results or the TLS result. This operation also provides flexibility in terms of the restrictions imposed on the TLS and IP-related data. For example, it may require only the presence of all IPv4-related results to pass an entry. This way, the operation can be fine-tuned in case of new findings on the effect of IP-related data on the classification. Should it be necessary to include results with other missing data, the filter operation can be adjusted and/or outer joins may be used.

The operation intentionally omits domain names for which the zone could not be found (*e.g.* dead domain names) in line with the requirements. If such entries were to be processed, a direct path would be inserted instead between the zone collector and the feature extractor.

An SQL query that describes the operation is provided in Appendix Section B.10. The relations ending with `Results` map to the state stores and the filter operation is represented by the `WHERE` clause at line 38.

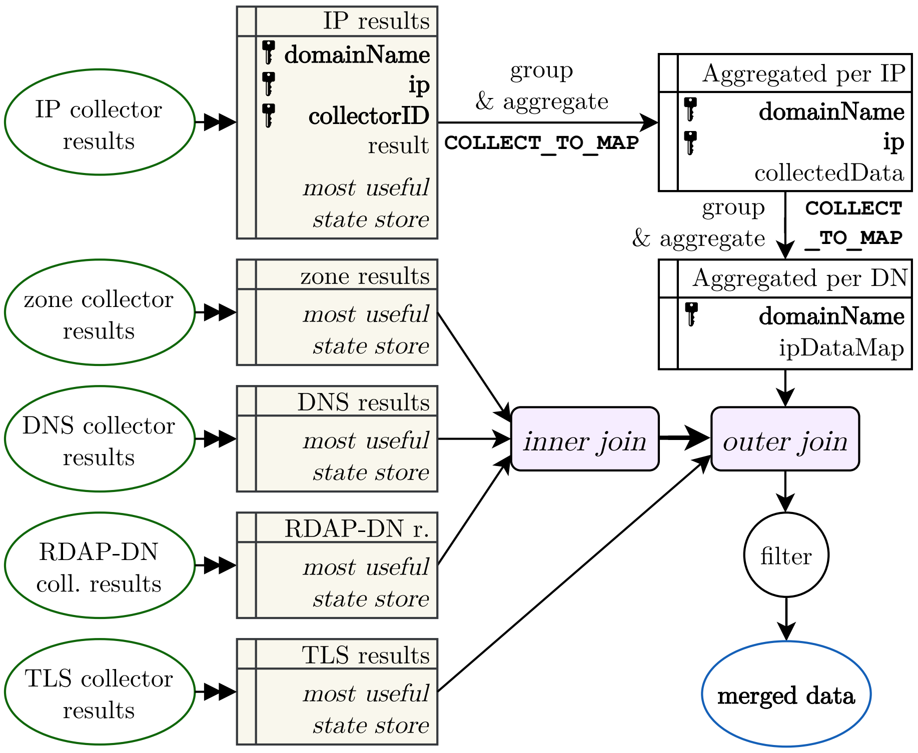


Figure 6.3: A scheme of the data merging operation. Note that the non-IP state stores have a domain name key attribute and a result attribute, which were omitted for clarity.

A note on processing guarantees

The nature of the data merging operation ensures that the rest of the system must only provide the at-least-once message processing guarantees (on the application level) to eventually initiate the feature extraction process for each input domain name. As long as each collector produces at least one result for every input, the data merger will eventually output the "most complete" data object for each domain name. Of course, to ensure the guarantees, the underlying platform must provide the at-least-once guarantees for the message delivery.

The guarantees will not be met if a pipeline component fails to produce a result. These cases can be mitigated by a re-collection controller that can check for missing results, accounting for when the other results were produced and for the typical time it takes to produce a result.

If a collector produces various outputs, there are two possible outcomes of the data merging process:

1. the new result is more useful than the previous one – the merger will produce a new, better result for the domain name;

2. the new result is less useful or the same – the merger will not produce a result.

In the first case, the only difference is that the feature extractor and the classifier will process the same domain name twice, so the rest of the system must be able to handle this situation. Specifically, the data storage mechanism must be able to handle multiple entries for the same domain name, and the user interface must be able to display only the relevant results. A potential issue is the waste of resources due to redundant processing.

6.6 Feature Extractor

The feature extractor stage consumes the unified data objects with collection results from the merger and produces feature vectors. The input and output channels are defined in Table 6.3. Note that an output feature must be a number, a boolean or `None`. This definition is based on the inputs expected by the existing classification module.

Input		
Key:	`str`	a domain name
Value:	`AllCollectedData`	the merged data object, see B.11
Main output (feature vector)		
Key:	`str`	the input domain name
Value:	`fvector`	the feature vector

Table 6.3: The input and output definitions of the feature extractor.

Internally, the feature extractor is defined as a list of arbitrary stateless transformations defined by a name, a list of features they create, and the transformation function itself. The model of a transformation is shown in Listing 6.2.

```
class Transformation:
    name: str
    features: dict[str, ftype]
    transform: (AllCollectedData, fvector) -> fvector
```

Listing 6.2: The model of a transformation. The values of the `ftype` type are the data types from the `feature` union. `transform` refers to a function with two parameters and one output.

The feature extractor works by gradually applying transformation functions to a feature vector value. The function has access to the entire original input data object, which is considered immutable. The available transformations are predefined by the implementation. The list of transformations, including the order of their execution, may be statically configurable but cannot be changed for a single input message. The feature extraction process is defined in Algorithm 6.1.

It is expected that the feature extractor will not offer any runtime configuration, as it does not make much sense to change the behaviour unless the classification model is changed – and in that case, the transformations should be updated as well. In case it does, the same configuration exchange mechanism as in the collectors is used.

Integration of the classifier unit

From the design point of view, the classifier unit is just another stage in the pipeline. It follows the same patterns as the other stages in the pipeline: It is an independent input/output block exchanging messages. It consumes the feature vectors from the feature

Algorithm 6.1: The feature extraction process

```
Input: domain name $dn$; AllCollectedData object $D$; list of transformations $T$
Output: domain name $dn$; feature vector $F$
1 $F \leftarrow$ new empty fvector
2 foreach $t$ in $T$ do
3     foreach feature key, feature type in t.features do
              /* note that each feature value may implicitly be None  */
4         $F$[feature key].expectedType $\leftarrow$ feature type | None
5         $F$[feature key].value $\leftarrow$ None
6     $F \leftarrow t.transform(D, F)$
7 if any value in $F$ does not match the expected type then
8     throw error
9 return $dn$, $F$
```

extractor, as defined above, and produces classification results. No output models are defined for the classifier unit, as it is out of the scope of this work.

6.7 Re-collection Controller

Algorithm 6.2 shows a very simple example of re-collection logic that could be used in a manually or periodically started re-collection controller. For each key/collector type pair, the controller keeps a number n denoting the number of retries. R are constants denoting the maximum number of retries for each collector type. The controller iterates over all failed results and requests re-collection for each of them. However, the effective meaning of certain results is that re-collection would likely not succeed – for example, if there is no RDAP endpoint for a TLD, it is improbable that it will appear any time soon. Thus, the controller may skip re-collection attempts for such results. Note that the controller should operate over a similar "most useful state store" as the merger, as it is only concerned with the last result that would be considered by the feature extractor.

Algorithm 6.2: An example of a simple re-collection logic

```
1 foreach failed zone result for dn with $n < R_{zone}$ older than $T_{zone}$ do
2     request zone collection for dn
3     $n \leftarrow n + 1$
  /* A DNS result may be considered failed if the "errors" dictionary
     is not empty.                                                  */
4 foreach failed DNS result for dn with $n < R_{dns}$ older than $T_{dns}$ do
5     request DNS collection for dn
6     $n \leftarrow n + 1$
  /* algorithm continues on the next page                           */
/* algorithm from the previous page continues                                  */
7  foreach failed TLS result for dn with $n < R_{tls}$ older than $T_{tls}$ do
8      if result status code $\neq$ CANNOT_FETCH then
9          request TLS collection for dn
10         $n \leftarrow n + 1$
11 foreach failed RDAP-DN result (both RDAP and WHOIS) for dn with
    $n < R_{rdapDN}$ older than $T_{rdapDN}$ do
12     if result status code $\notin$ { NO_ENDPOINT, NOT_FOUND } then
13         request RDAP-DN collection dn
14         $n \leftarrow n + 1$
15 foreach failed RDAP-IP result for dn/ip pair with $n < R_{rdapIP}$ older than $T_{rdapIP}$
    do
16     if result status code $\notin$ { NO_ENDPOINT, NOT_FOUND } then
17         request RDAP-IP collection for dn/ip
18         $n \leftarrow n + 1$
19 foreach failed NERD result for dn/ip pair with $n < R_{nerd}$ older than $T_{nerd}$ do
20     if result status code $\neq$ UNSUPPORTED_ADDRESS then
21         request NERD collection for dn/ip
22         $n \leftarrow n + 1$
```

Chapter 7

Implementation

This chapter describes the implementation of the data processing pipeline and its integration with database systems. The pipeline is based on Apache Kafka. Each stage is an independently running program (a microservice), and Kafka transfers messages between them. Commands and processing requests, collected data responses, feature vectors, or logs materialise as Kafka events. Kafka's partitioning and consumer group management ensure scalability by distributing the work across multiple instances of the stages. They can also be used, together with Kafka's replication mechanisms, to ensure the system's fault tolerance.

The pipeline stages are implemented using some of the platforms mentioned in Section 3.5. Several collectors and the feature extractor are written in Python using the Faust framework, mainly for compatibility with existing code and because it was easier with respect to the available libraries. Some collectors use Java instead, leveraging the Confluent Parallel Consumer library for a highly parallelised collection that is more effective than the Python-based solution. The data merger is based on Kafka Streams (KS). The Kafka Connect (KC) platform integrates the pipeline with PostgreSQL, which stores the domain names to collect, and with MongoDB, where the collected data are persisted.

Section 7.1 explains why the mentioned technologies were chosen and how they connect to form the DomainRadar system. Section 7.2 presents the Kafka-based data processing pipeline, showing how the individual components connect. Section 7.3 gives details of the implementation of the Python-based collectors, followed by Section 7.4 that describes the feature extractor, which is based on the same framework. Section 7.5 describes the Java-based collectors and the pipeline component that merges the collected data. Section 7.6 explains how KC provides input to the pipeline and store the collection results in the database systems. Section 7.7 presents the included Docker images and Compose files that allow the system to be easily deployed in local scenarios.

7.1 The Big Picture: the Technologies of DomainRadar

Figure 7.1 shows an implementation look at the DomainRadar architecture proposed in Section 5.1. The scope of the work in this thesis is the distributed data processing pipeline, shown in the yellow block. The bold lines show the most common path taken by newly seen domain names to reach the collection process: They are ingested from Elasticsearch by the loader that, if they pass the filters, saves them to a PostgreSQL database. New entries found in the database are imported via KC. Then, each entry passes through the collectors, the feature extractor, and the classifier unit[1]. All collection (and classification) results are sent back to Kafka, and KC saves selected fields to the two databases:

PostgreSQL[2] is an open-source (object-)relational DBMS [119]. It provides fast storage for the input loader and stores metadata on the results of the collection attempts that can be used in the re-collection controller. It supports embedding JSON objects in entities, which is used to store dynamic extra information from the loader.

MongoDB[3] is a general-purpose document database with support for sharding and replication. It offers an expressive query API that provides complex aggregations inside the DBMS [86]. It stores the collected data and classification results, as it is well-suited for storing complex data structures. The aggregation framework is used to transform the stored data into any target form, *e.g.* the data structure expected by the UI.

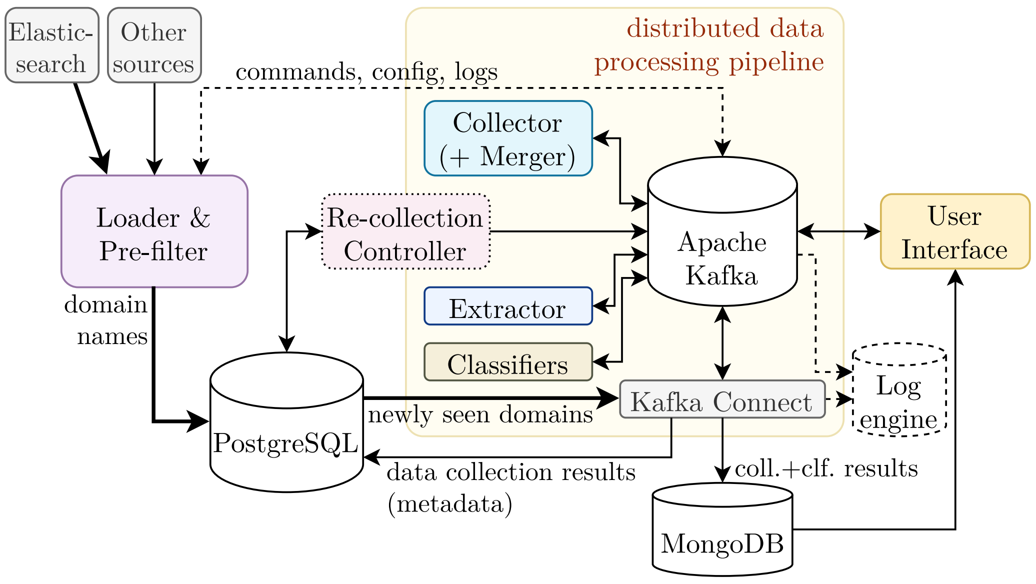


Figure 7.1: A scheme of the DomainRadar implementation architecture. The bolder arrows show how newly seen domain names reach the data processing pipeline. Dashed lines represent operational data, such as logs and commands.

Loader & Pre-filter: The loader uses the "upsert" (update or insert) operation to store all the DNs that pass the filters, regardless of whether they have already been stored before. This operation inserts previously unseen DNs, assigning them unique IDs from a sequence and a "first seen" timestamp. For DNs already present in the database, it only updates the "last seen" timestamp. This is the primary reason for introducing of PostgreSQL, as it can perform this operation efficiently even for large batches of entries.

Based on this behaviour, there is a notable difference to the proposed design – the absence of a direct link between the loader and the collector. While the loader could determine which domains are "new" by comparing the timestamps, it does not make much sense to concern it with the task of passing data to the collectors or determining which DNs

[1]The classifier unit is considered a part of the distributed data pipeline, but it is not a part of the implementation described in this work.

[2]PostgreSQL: https://www.postgresql.org/

[3]MongoDB: https://www.mongodb.com/

should be re-evaluated. Instead, with information on domain names to process already in the database, it is cleaner to use the unified and optimised data exchange capabilities of KC. Its PostgreSQL connector keeps track of the last ingested ID and imports the new entries to a “to process” topic in Kafka. If re-evaluation of DNs processed in the past is required, the re-collection controller can use the timestamps to decide on the target DNs.

Re-collection Controller: The controller reads information on past collection attempts and their results from the PostgreSQL database and decides which domain names should be re-collected. The controller crafts the request messages for the target collectors and publishes them to Kafka. Note that it is shown on the boundary of the distributed pipeline: While some implementations may leverage the system’s distribution model, it is expected that initial implementations will work in a single instance. The controller will be addressed in future work.

UI: The user interface is a web application that presents the classification results for the individual domain names. Users can also use it to edit the system’s dynamic configuration. It uses a simple backend that loads data exclusively from the MongoDB database. An example of an aggregation pipeline that transforms the stored data into a presentation form is included in the attached storage. The UI also connects to Kafka to exchange operational messages, such as the configuration.

Logging: The scheme includes a dashed database-like entity that represents a log analytics system. The implemented components push the logs to a Kafka topic from which this external system may consume them. Alternatively, the external system could be connected through KC. This is an infrastructure detail and is not further elaborated on in this work.

Modifications for standalone collection

Figure 7.2 shows how the implementation architecture can be modified when the data processing pipeline is used outside of the DomainRadar system, typically for the purpose of collecting new training datasets. Here, the system is simplified considerably. The loader is replaced by a simple input controller that reads the domain names from a source (such as a text file) and sends collection requests to the collectors. In the future, the collector will be extended to provide the re-collection logic.

For model training, it is also necessary to include the feature extraction process. Figure 7.2b shows this modification. The feature vectors may be stored in MongoDB or in a different system – for example, written in a binary serialised form to files in the filesystem.

Both the collected data objects and the feature vectors are first stored in Kafka and then transferred to MongoDB through Kafka Connect. This way, the pipeline stages are not concerned with data persistence at all. The storage can be easily changed by using a different KC connector.

On the selection of Apache Kafka

Apache Kafka was chosen as the base platform for the implementation of the DomainRadar system and its data processing pipeline because it naturally fits the system’s data-flow model and requirements, enabling high-throughput message exchange and distribution. The main motivations were the flexibility of implementation platforms, as well as their deployment

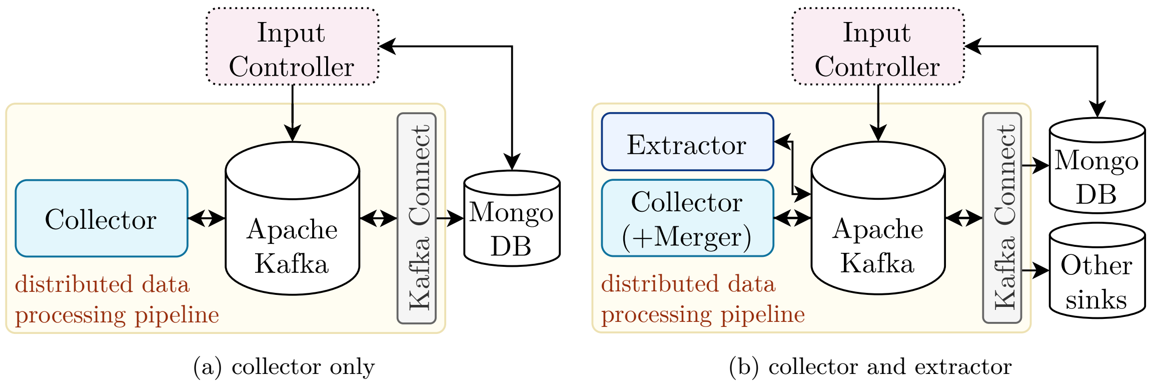


Figure 7.2: Two schemes showing how the system architecture may be simplified for standalone data collection (and extraction). Case (a) may be used when only the raw data are required. Case (b) includes the feature extraction process. The re-collection logic is included in the input controller.

and scalability. The system is built as a set of independent components exchanging data through Kafka. This way, it fulfils various requirements:

Scalability The system can be easily scaled horizontally by adjusting the partitioning and adding more instances of the individual components. Importantly, the number of instances of each component can be chosen independently based on the observed traffic and behaviour.

Deployment flexibility Almost any deployment topology may be chosen: the components may run on a single machine, in containers, on a set of virtual machines, or even in a cloud environment. It also fulfils the requirement of deploying certain components in geographically separate locations.

Unified data exchange Kafka can be used both to transfer the actual working data between the components and to exchange system data, such as logs or commands. It can also be used as persistent storage for some of the data, such as the configuration.

Security The components may be deployed in a fairly isolated environment: in addition to the target external services, they only need to be able to connect to the Kafka cluster. Their communication can be encrypted, and strong authentication between all entities in the system can be enforced with little effort.

Maintenance It is easy to replace or upgrade the independently running components without affecting the rest of the system. Furthermore, Kafka provides embedded mechanisms for monitoring.

Reusing existing code As Kafka clients are available in many programming languages, parts of the codebase used in the preceding research could be used when developing the pipeline components.

Integration with other platforms Kafka Connect is a well-tested platform that allows easy integration with various database systems. Also, many data processing platforms, such as Apache Flink, provide connectors to Kafka, which may prove useful for future extensions of the system.

Efficiency The performance of the Kafka cluster affects the overall efficiency of the system. It can be tuned by adding more nodes or changing the configuration. Kafka is successfully used in diverse scenarios by many large companies, supporting the claims of its performance and reliability [13]. The pipeline components are lightweight and designed so that the individual instances do not require much system resources.

Replication and fault tolerance are provided by Kafka itself.

Other platforms were considered, but large-scale data processing systems like Apache Flink and Apache Spark do not fit well with the DomainRadar pipeline, which focuses on a particular way of data enrichment. Flink's support for asynchronous I/O is not its primary use case and lacks the required deployment flexibility, as the framework controls computation locations rather than the user. In the future, these platforms might be used in DomainRadar for tasks like feature extraction or training new models, where they excel. Using Kafka also has drawbacks, primarily the need for system maintenance and monitoring. Setting up a basic Kafka cluster is straightforward, but fine-tuning its configuration for maximum performance, reliability, and efficiency, especially in large-scale clusters, is complex. Cloud-based solutions like Confluent or Redpanda Cloud could mitigate some of these issues. Additionally, Kafka's scalability is limited by network bandwidth, disk I/O, and managing many partitions, which can become bottlenecks.

7.2 The Kafka-based Data Processing Pipeline

The individual collectors, the data merger, and the feature extractor are implemented as standalone microservices that communicate through Kafka. Their designed input and output channels correspond to topics from which the components consume and to which they produce messages. The design from Figure 6.2a is materialised in the implementation shown in Figure 7.3. Each collector has one primary input topic, prefixed with *to_process*, and one primary output topic for the results, prefixed with *processed*. The *collected_IP_data* topic serves as the output bus for all IP-based collectors. Some collectors additionally publish messages to the *to_process* topics of other collectors – this way, a pipeline is formed. The figure includes the types of keys and values of the exchanged messages, which correspond to the models specified in Appendix Section B.11. All components are assigned a component ID, and additionally, collector components are assigned collector IDs. See Appendix C for the assignments.

Serialisation

In the current version of the implementation, all complex types in the system are serialised to JSON [44] and stored in Kafka, encoded as UTF-8 byte sequences. The values are in a compact format: they contain no extra whitespace characters or newlines. The JSON format was chosen for its simplicity and human-readability, which is helpful in development.

Most types from Table 6.1 have direct counterparts in JSON. The exceptions are:

- `timestamp`: serialised as an `int` representing the number of milliseconds since the Unix epoch,
- `ip`: serialised as a `string` that contains an IPv4 or IPv6 address in the standard notation,

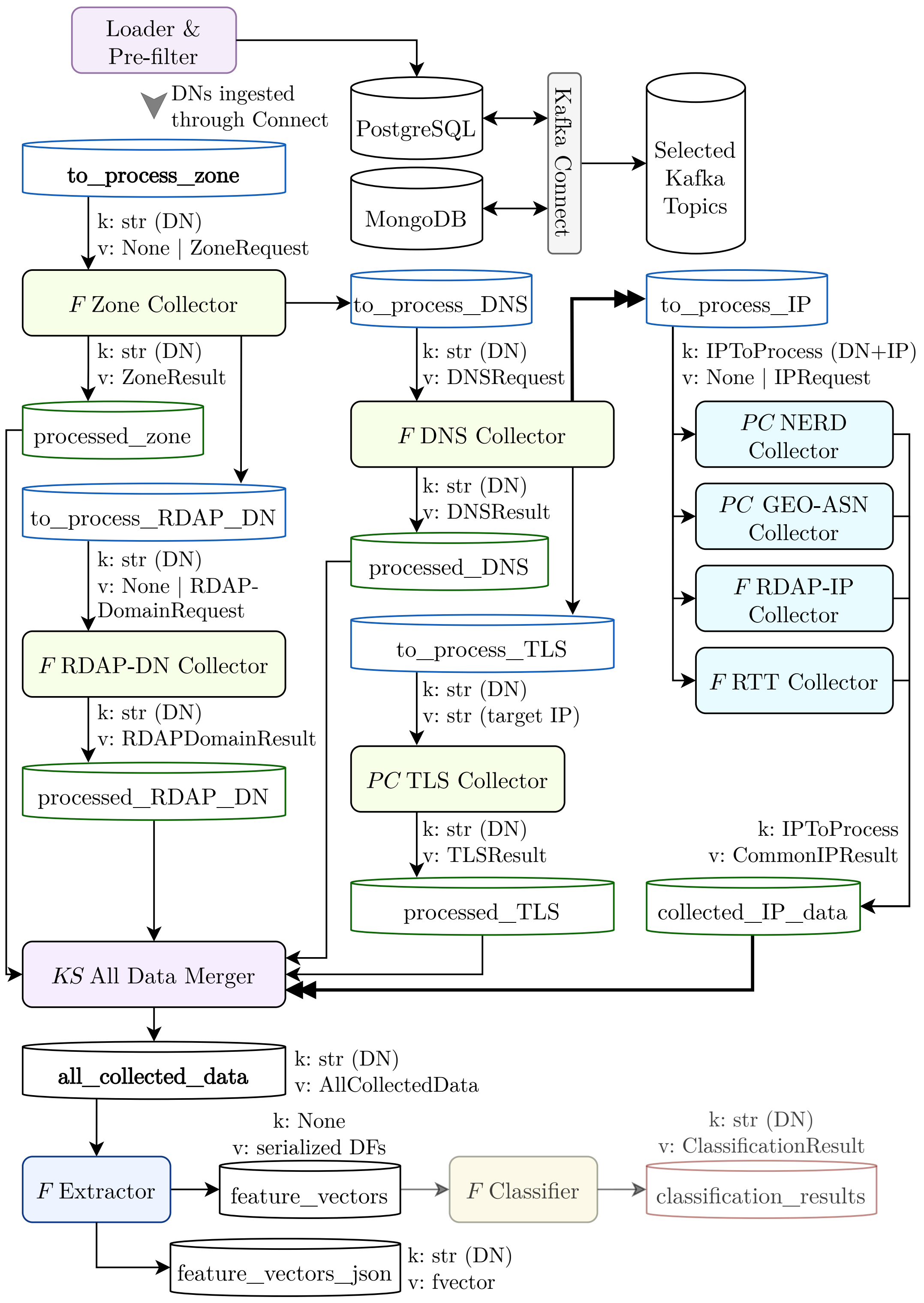


Figure 7.3: A detailed look on the Kafka-based distributed data processing pipeline from Figure 7.1. The double arrows mark that multiple records are produced or consumed per a single input DN. The *F* mark signifies a component based on Faust while *PC* stands for Parallel Consumer. The classifier unit is also indicated for completeness.

- `bytes`: serialised as a `string` that contains the data in the Base64 encoding.

Note that neither JSON nor Python imposes any particular limit on the size of the numbers. Additional restrictions apply to ensure compatibility with the Java components:

- `int` is a 64bit signed integer,
- `float` is a floating-point number representable in the IEEE 754 binary64 format.

Configuration management

The design specification requires that the components should offer an input channel for runtime configuration exchange. Kafka is used to distribute and store the configuration:

- the *configuration_change_requests* topic is used to send configuration changes to the components,
- the compacted *configuration_states* topic is used to send responses and persist the current configuration.

Component IDs are used as the keys for the events in these topics.

Work on the component implementations started before the configuration exchange system was fully specified, so they were designed to read all their configuration from a file at startup. Instead of reworking the components, a standalone configuration manager is provided, which works in conjunction with the Compose-based service orchestration described in Section 7.8. It is a simple Python program that runs as a daemon. It consumes requests from the Kafka topic, applies the changes to the configuration files, and restarts the running services. It also publishes the current configuration to the *configuration_states* topic.

The manager ensures that the important static properties, such as Kafka connection settings, cannot be changed at runtime. It reads the existing configuration files, and when a change request is received, it copies the static properties from them. The static properties are also excluded from the `currentConfig` configuration snapshot in the result messages. If a static property is included in the input object and does not match the current value, a soft error with the error code `READ_ONLY` is produced, but the input object is still applied. However, the manager does not currently perform any other validation of the input configuration objects.

The source code of the manager is included in the `/src/config_manager` directory. It is based on the confluent-kafka-python[4] client library. The `README.md` file contains a how-to-use guide and a description of the sources. In the future, the components could be modified to participate in the configuration exchange mechanism directly, obsoleting this external manager.

Topic settings and initialisation

The `/infra/kafka_scripts/prepare_topics.sh` script is provided to create and update the necessary Kafka topics. At the top, the script contains definitions of topics, their number of partitions, and per-topic configuration, such as the cleanup policy. The script connects to Kafka and creates the topics if they do not exist. The `UPDATE_EXISTING_TOPICS`

[4]confluent-kafka-python: https://github.com/confluentinc/confluent-kafka-python

environment variable controls if the script also updates the configuration of existing topics, and the `UPDATE_PARTITIONING` variable controls if it alters the number of partitions.

The script contains four variables: `COLLECTOR_`, `MERGER_`, `EXTRACTOR_`, and `CLASSIFIER _PARALLELISM`. They control the number of partitions of the respective input topics, affecting the maximum number of component instances that can process data in parallel.

In addition to the topics shown in Figure 7.3, additional topics are created:

- *filtered_input_domains* is used for transferring certain records from the loader to the MongoDB database (through KC),
- *connect_errors* is used as the "dead letter queue" where KC sends messages that could not be processed,
- *configuration_change_requests* and *configuration__states* are used for the configuration exchange mechanism.

The *delete* policy mandates that the messages are removed at a specified time after they are published. It is used for:

- all the *to_process_* and *processed_* topics, as well as for *collected_IP_data* (48 hours),
- *filtered_input_domains*, *feature_vectors*, *feature_vectors_json*, *configuration_change_requests* (1 hour),
- *connect_errors*, *component_logs* (7 days).

The lifespan of the inter-collector messages was chosen as a very conservative estimate of the maximum time it should take for a message to be processed by all collectors. It also defines the maximum time in which the re-collection controller does not have to re-introduce past collected data to Kafka. The 1-hour lifespans were chosen for transient messages that will be consumed quickly by KC or the target components. The 7-hours lifespans were chosen for the general logging topic and the dead letter queue so that the operator has time to notice the errors.

The *compact* policy mandates that only the latest message is stored for each key, while the previous are eventually removed. It is used for:

- *all_collected_data*, *classification_results* (min lag: 1 hour, max lag: 12 hours),
- *configuration_states* (min lag: 10 minutes, max lag: 1 hour).

The merged data objects and the classification results are currently kept compacted mainly for debugging purposes. Normally, the data objects can be reconstructed from the collection results stored in MongoDB, and classification results are also stored in MongoDB. The configuration states are compacted quickly, as only the last message for each collector should always be used.

Co-partitioning

In Kafka Streams, the *co-partitioning* requirement is essential for ensuring that join operations between two or more streams or tables are executed correctly. Co-partitioning mandates that all involved streams (or tables) must share the same number of partitions and that their data are written using the same partitioning strategy [12]. For example, when joining the data from the DNS and the TLS collectors by the key, if a DNS record is in partition P, the corresponding TLS record must also be in partition P. Without co-partitioning, the join operation would produce incorrect results. For this reason, all the *processed_* topics must use the same number of partitions. The *collected_IP_data* may be partitioned differently, as it uses a different kind of key (the DN/IP pair) and is re-partitioned by KS during the aggregations.

Standalone input controller

Included with this work is an example of an input controller that enables standalone collection and feature extraction outside of the DomainRadar system (for example, for creating new large datasets). It is a simple Python program with a command-line interface that reads domain names from a text file (local or remote) or from a MISP feed and sends them to Kafka for processing. Collection and extraction results are saved to a MongoDB database through KC. The program also connects to MongoDB in order to store metadata on previously loaded DNs so that no entries are processed twice (unless explicitly requested).

The source code of the input controller is included in the `/src/standalone_input` directory. The `README.md` file contains a how-to-use guide and a description of the sources. The program is based on the confluent-kafka-python client library and uses PyMongo[5] to interact with MongoDB. The code for loading domain names from various sources and, partially, the database access code were adopted from [57].

7.3 Python-based Collectors

The source code of the Python-based pipeline components (the collectors, the feature extractor) is included in the `/src/python_pipeline` directory. The Python projects use Poetry[6] for dependency management – the dependencies are set in the `pyproject.toml` project files. There are two independent projects in the `collector` and `extractor` directories with their own project files, although both projects include the `common` module with shared code. The top-level "meta-project" file only links the two files as local dependencies and is provided for easier management of the development environment.

Each Python-based collector is implemented in a separate namespace (directory) and contains its own `__main__.py` so that each collector can be executed using the `python -m <namespace package>` command. These entrypoint scripts ensure that the data directory required by Faust is present, and they delegate to Faust's umbrella command-line interface[7] so all Faust commands may be used with them. For example, a collector may be executed using `python -m collectors.zone worker`.

Each Faust application must be configured with an application ID. It is mainly used as the Kafka consumer group ID, correlating all instances of the same application. By default, the application ID is set to `domrad-[collector ID]`. It can be overridden using the `app_id` configuration property.

[5]PyMongo: https://pymongo.readthedocs.io/
[6]Poetry: https://python-poetry.org/
[7]Faust CLI: https://faust-streaming.github.io/faust/userguide/cli.html

Data models

Implementations of the data models from Appendix Section B.11 are in the `common.models` module. The Pydantic library[8] is used for model management. Models are defined using Python dataclass syntax. The library can convert between JSON representations and the model classes while performing validation of the input data. Faust has its own model management system, but it was found unsuitable. Instead, the `common.custom_codecs` module defines two custom Faust *codecs*, classes used for value (de)serialisation: `StringCodec` and `PydanticCodec`. The latter ensures interoperability between Faust and the Pydantic-based models.

Configuration

The collectors use static configuration files loaded from the filesystem. Configuration is defined using the TOML[9] format. It is directly representable as a Python dictionary and a TOML deserialiser is included in Python's standard library, making it easy to load and work with. For each collector, the configuration contains the Kafka connection and security settings, the Faust application settings, logging settings, and collector-specific settings. An example configuration file with all available properties and their descriptions is available in `config.example.toml`.

Configuration mapping

The configuration manager provides mapping between the TOML format used in the configuration files and the JSON format used by the runtime configuration exchange:

- When mapping a saved TOML file, the tomllib module[10] from Python deserialises the file into a dictionary and the `json.dumps` function[11] serialises it to a JSON string.
- When mapping an exchange JSON string to the TOML format, the `json.loads` function deserialises the string into a dictionary and the tomli-w library[12] serialises it to a TOML string.

Note that the manager produces an `INVALID_TYPE` soft error for any fields with `null` value as it is not supported in TOML. The fields will be omitted from the target TOML file.

In all Python-based collectors, the static configuration (omitted by the manager) includes:

- the entire `connection` section (including subsections) that controls the connection to Kafka,
- all fields named `app_id` (in any section).

An example of a mapping is shown in Listing E.1.

[8]Pydantic library: `https://docs.pydantic.dev/`
[9]TOML: `https://toml.io/`
[10]tomllib module: `https://docs.python.org/3/library/tomllib.html`
[11]json module: `https://docs.python.org/3/library/json.html`
[12]tomli-w library: `https://github.com/hukkin/tomli-w`

Logging

A shared logging wrapper is implemented in the `common.log` module. It uses Python's logging facility[13] to create customised loggers for the components. A `TRACE` log level is added for in-depth execution logs. Custom logging functions are injected in the `Logger` class to provide a concise interface. They include a `key` parameter that inserts the currently processed entry key to the log record using a custom formatter. A custom log handler is included that writes logs to Kafka, independent of the Faust application. Several configuration options are available to define the minimum logging levels.

The general structure of a Faust-based collector

All collectors share a common structure of the main module shown in Listing D.1. In general, the collector is a simple module that:

1. initialises the logging system and reads the configuration,
2. initialises the Faust application,
3. defines a single input topic and one or more output topics, and
4. defines a single Faust agent for the input topic.

The Faust agent is a function that asynchronously consumes incoming events from the input topic, deserialises the value of the event, executes an async collecting operation, and produces its results to the output topic(s). The processing of each entry is encapsulated in a try-except block that, in case of an unexpected error, logs the error and sends an `INTERNAL_ERROR` result so that the entry does not appear unprocessed to the next stages. The structure is similar for IP-based collectors. They must additionally deserialise the event's key and, when the event has a value, check if it requests this collector to run. The modified structure is shown in Listing D.2.

The agents in all collectors set the `concurrency` parameter[14] based on the configuration. When set to $n > 1$, the agent will internally execute the incoming events in n independent tasks. This does not mean they will run in n real threads but rather that the event loop will be able to switch between the n tasks, and no particular order will be enforced on the order of processed elements. It is useful for collectors that perform I/O-bound operations, such as network requests, as it allows the collector to process multiple events concurrently.

DNS collector

The DNS collector (`collectors.dns`) uses the dnspython library[15] to perform DNS queries. It can directly build DNS queries and send them to specified nameservers, but it also provides a stub DNS resolver[16] that provides DNS recursion using a remote full resolver. For all APIs, the library provides both synchronous and asynchronous versions of the functions, which can be leveraged by the asyncio-based collectors. For each DNS query, the collector first tries to use the provided list of primary nameserver IPs. If a query to a nameserver

[13]logging facility: https://docs.python.org/3/library/logging.html

[14]Concurrency (Faust docum.): https://faust-streaming.github.io/faust/userguide/agents.html#concurrency

[15]dnspython library: https://www.dnspython.org/

[16]Stub DNS resolver: https://dnspython.readthedocs.io/en/stable/resolver.html

fails, the address is removed from the list, and if no addresses are left, it falls back to using the stub resolver with the globally configured DNS recursive resolvers.

Figure 7.4 shows a simplified time diagram drafting how the collector operates. The program is based on a loop executing events from a runtime-managed queue. It accepts a domain name and starts an asynchronous DNS query. This operation stores the program context and the "execution flow" is suspended until the query completes. In the meantime, the program may work on the next item from the queue (*e.g.* it can accept another domain name). When a query completes, its result and the context are inserted as an event into the queue. The program eventually jumps back to this context and proceeds by triggering another query. This ensures that multiple queries are processed in parallel, but for each DN, the implementation appears sequential to the programmer. This process is effective when the collector is saturated, *i.e.* it always has domain names to process.

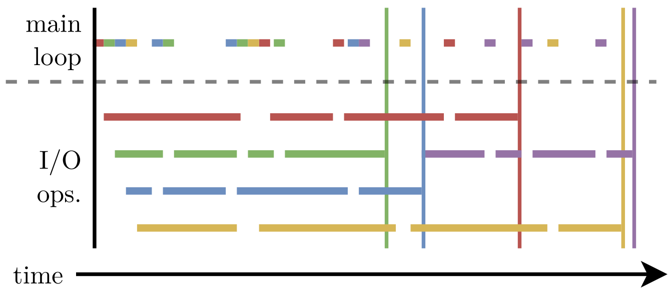


Figure 7.4: An event-loop based implementation of the DNS collector. Each colour represents a single processed domain name. Each long horizontal line represents waiting for a DNS response to a single query. Vertical lines show when the processing finished for a single domain name.

Zone collector

The zone collector (`collectors.zone`) also uses the dnspython library. Here, only the stub resolver is used to make the necessary queries using the configured recursive DNS resolvers. The tldextract library[17] determines the public suffixes of domain names. The collector works similarly to the DNS collector. Its options include the target DNS resolvers, the timeout for a single query and the round-robin target resolver selection option.

RDAP-DN collector

The RDAP-DN collector (`collectors.rdap_dn`) uses the whodap library[18] to perform RDAP queries and the asyncwhois library[19] to perform WHOIS queries. The tldextract library is again used to determine public suffixes (when trying the possible registered name).

The asyncwhois library automatically parses the WHOIS responses using an embedded parser. In the current implementation, the output parsed dictionary is added to the `RDAPDomainResult` model as an extra `whoisParsed` field. However, it may be removed in the future, as it is the feature extractor's responsibility to extract the features from the WHOIS data.

[17]tldextract library: https://pypi.org/project/tldextract/
[18]whodap library: https://github.com/pogzyb/whodap
[19]asyncwhois library: https://github.com/pogzyb/asyncwhois

The asynciolimiter library[20] provides the local rate-limiting capability. The library implements three subtly different algorithms for rate limiting based on maximum rate per second that differ mainly in burst handling [52]:

- `Limiter` does not pass any bursts unless delayed by a long-running CPU task that caused the limiter not to be checked in time,
- `StrictLimiter` never passes bursts, so the resulting rate is always bounded by the specified one,
- `LeakyBucketLimiter` implements the leaky bucket (as a meter) algorithm. It allows bursts up to the specified capacity but the request rate is bounded.

Users can configure the used rate limiter and its options globally and per RDAP endpoint or per TLD. The `collectors.limiter` module manages the limiter instances for the individual endpoints. It also handles the immediate and queueing modes, wrapping the asynciolimiter's `wait` function with custom timeouts. It provides a function that forces the limiter's capacity to be exhausted, which is used to prevent the collector from sending more requests when a rate-limiting error is received from the remote endpoint.

Some RDAP servers were found to use deprecated TLS ciphersuites. For this reason, the collector creates a custom SSL/TLS context with the `ALL:@SECLEVEL=1` parameter that decreases the required security level to 1, which corresponds to a minimum of 80 bits of security [97]. In the future, this should be made adjustable per endpoint.

RDAP-IP collector

The RDAP-IP collector (`collectors.rdap_ip`) is implemented similarly to the RDAP-DN collector, using the whodap library and the same rate limiter infrastructure. Configuration properties are also the same.

RTT collector

The RTT collector (`collectors.rtt`) uses the icmplib library[21] to send and receive the ICMP Echo/Echo Reply messages. It implements the necessary ICMP protocol functions directly in Python without the need for external dependencies. It provides a straightforward asynchronous API for performing the pings that returns an object with the required statistics of the completed operation.

On Linux, root privileges or the `CAP_NET_RAW` capability are required to send an ICMP message, as they can only be sent through a raw socket. However, since 2011, there has been an extra socket type that can be used to send an ICMP Echo message (and receive the Echo Reply) specifically [69], and in 2013, it was extended to IPv6 [32]. This is done by using a `SOCK_DGRAM` set to the `ICMP_PROTO` protocol type. The library can work both in an "unprivileged" mode, leveraging this feature (if supported by the OS), and in a "privileged" mode, using raw sockets. The collector can be configured to use either mode. The only other available setting is the number of ping messages to send.

[20] asynciolimiter library: https://asynciolimiter.readthedocs.io/
[21] icmplib library: https://github.com/ValentinBELYN/icmplib

7.4 The Feature Extractor

The feature extractor is another component implemented in Python using the Faust framework. It is located in the `/src/python_pipeline/extractor` directory and shares the `common` module with the collectors. The general information stated in the previous section applies to the extractor as well. The transformation functions were largely adopted, revised, and integrated from the DomainRadar experimental code[22], originally developed by Horák in [57]. The table in Appendix F lists all the features currently computed by the feature extractor. The original feature extraction code and the feature descriptions were written as a part of the DomainRadar project in cooperation with R. Hranický, A. Horák and J. Polišenský.

Batched processing using pandas

Horák's feature extraction system was designed for one-off extraction of large bounded datasets. It represented the data using pandas[23] data frames. Pandas is a data analysis and manipulation library for Python that provides highly optimised, efficient operations over tabular datasets [93]. A data frame is a two-dimensional, mutable, indexed data container.

All input data objects were key-value mappings with the same uniform structure (set of keys). The extractor loaded the objects from a database into a single data frame, where the keys defined columns and each object was represented as a row. The data frame was then processed by the transformation functions, which were implemented as operations over the table. The resulting data frame was serialised into a binary format. When used properly, this approach can be quite efficient, as the pandas operations are vectorised and executed in native code.

The implementation of the feature extractor presented here adopts this approach of processing batches of data using pandas data frames. To embrace the optimisations offered by pandas, it does not process the incoming data one by one. Instead, the `take_events` function provided by Faust gathers up incoming events into a batch of a configured size. The data frame is then created from all the events in the batch. The transformation functions are applied one by one to the data frame. The entire resulting data frame is serialised into a binary format and published to the output topic, from which the classifier unit reads the data and classifies all the objects in the batch at once. The extractor can also be configured to output the feature vectors one by one, serialised in JSON. Note that a batch may also be passed to processing after a configured time has passed since the first received event. The effects of adjusting the batch size are evaluated in Section 8.3.

Compatibility with the original code

The transformation process adopted from [57] differs from the design defined in Section 6.6 in that it does not pass the collected data object as an immutable side input to the transformations but rather initially converts the data objects to a data frame and mutates it. For ease of transition, the implementation follows this pattern, so the `transform` function is declared as `(DataFrame) -> DataFrame`.

[22]DomainRadar "playground": https://github.com/Ovi0vocny/dr-playground/, the adopted transformation functions are in the `transformers` directory.

[23]pandas: https://pandas.pydata.org/

The transformations were designed to work with data objects produced by the prototype data collection tool. Although it collected similar data, their structure was largely different to the one defined in this work. For this reason, the extractor includes a special "compatibility transformation" that accepts an `AllCollectedData`-like dictionary and uses it to create a new dictionary that reflects the legacy data object structure. The transformation mostly re-shapes the data, although it also handles the extraction of RDAP and WHOIS data, which was previously done by Horák's collector. The transformation is implemented in the `extractor.compat` module.

Operation

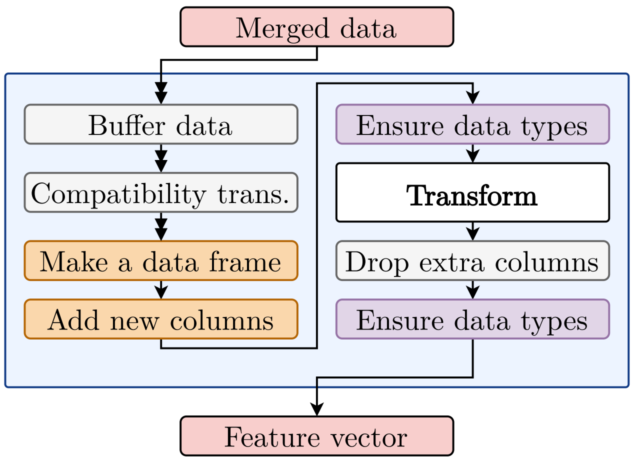


Figure 7.5: The implementation of the feature extractor. The white block includes the sequential chain of transformations shown in Figure 6.2b. Error handling is omitted.

Figure 7.5 summarises the extraction process implemented in the `extractor.extractor` module. The input to the `extract_features` function is the batch of deserialised input data dictionaries. Note that the main loop calling the function appends the original event's key (the domain name) into the dictionary.

1. All entries in the batch are processed using the compatibility transformation.
2. A data frame is created from the processed dictionaries.
3. It is extended with new columns defined by the transformations.
4. Correct data types must be set for the columns, as they may not always be correctly inferred from the input data.
5. The transformations are executed one by one. The order is specified by the module's `_all_transformations` field.
6. The final "drop" transformation removes the columns that are not needed in the output, including the columns with the original data.
7. The predefined data types are again enforced to ensure that the feature vector matches the structure expected by the classifier unit. Should any transformation produce incorrect types, this step would raise an error for the entire batch.

Transformations

All transformations (excluding the compatibility one) are defined in the `transformations` directory. The base abstract class `Transformation` is defined in the `base_transformation` module.A transformation class may accept the configuration dictionary in its constructorInheritors implement the `transform` method that mutates the data frame of feature vectors and the `features` property getter that returns a dictionary where keys are the output feature names and values are the string aliases of pandas data types[24].

To extend the system, one must create a new transformation class that inherits from the base and implements the two methods. The class object must be appended to the `_all_transformations` dictionary in the `extractor` module, keyed with an identifier for the new transformation. If the inheriting class has a constructor, it must contain exactly one positional parameter (the configuration dictionary).

Serialisation

The input key-value pairs are expected to be serialised in the same way as in the collectors. The feature extractor deserialises the input JSON string into a Python dictionary that reflects the structure of the `AllCollectedData` model.

For use in the DomainRadar system, the output data frame is serialised using the Feather V2[25] format, a binary format for storing data frames included in the Apache Arrow project. It was selected for its good speed according to the pandas documentation [94]. Events produced by the extractor have no keys; the source domain names are included in a column of the data frame.

For debugging and use outside of DomainRadar, *e.g.* for standalone collection, the feature extractor can also be configured to serialise the feature vectors as JSON objects and produce them as individual messages. In this case, the serialisation follows the same rules as in the collectors.

Configuration

Configuration works similarly to the collectors. The extractor-specific configuration is stored in the `extractor` section. The configuration includes the batch size and batch fill timeout, the options for producing JSON results, the enabled transformations and the number of workers (see below).

Error handling

When an error occurs during the compatibility transformation, the extractor logs the error and omits the entry from the data frame. However, when an error occurs inside one of the common transformations, it logs the error and discards the entire batch. For this reason, the transformations should internally handle all possible errors gracefully, *e.g.* by setting the feature to `None`.

Internal parallelisation and the ineffectivity of the transformations

While batched processing can increase the extractor's throughput in theory, it requires the transformations to be implemented in a way that allows them to be vectorised by pandas.

[24]pandas data types: https://pandas.pydata.org/docs/user_guide/basics.html#basics-dtypes
[25]Feather V2: https://arrow.apache.org/docs/python/feather.html

Unfortunately, the transformations from the research are not designed this way: they are mostly based on pandas' `apply` method[26], which applies a given function to each row sequentially, and thus it does not add any performance benefits when compared to manual iteration.

Due to Python's *Global Interpreter Lock* (GIL), only a single thread can execute Python code at once [107], so the apply function cannot be trivially parallelised through threading. Several libraries exist that extend pandas with parallelism based on multi-processing: they split the input data frame into chunks, distribute them to child processes, run the user function, and collect the results back to the main process. However, this approach adds overhead. Furthermore, it is also not suitable for the current implementation, as the transformations execute the `apply` function separately for computing each feature. The demanding process of splitting the data frame and collecting the results would be repeated for each feature, leading to infeasible overhead.

Pandas also allows running the apply method using the *numba* engine, which can provide a significant speedup by JIT-compiling the function and running it outside of the GIL. This could be used to speed up the transformations, but it would still require an overhaul of the implementation, as it requires the function to work with raw numpy arrays instead of the abstraction from pandas.

The implementation tries to mitigate the issue by the explicit use of Python's multiprocessing features. When `computation_workers` is set to a value greater than 1, the extractor creates a pool of worker processes. The main process collects and deserialises batches of input messages and then distributes them across the workers that execute the pandas operations. When a worker produces the final data frame, it is passed back to the main process, which serialises it and sends it to the output topic. The `worker_spawn_method` property can be used to change between the process spawn modes (spawn, fork, forkserver) offered by Python; see [106] for more information.

7.5 Java-based Collectors and the Data Merger

The source code of the Java-based pipeline components (the collectors, the data merger, and the KC extensions) is included in the `/src/java_pipeline` directory. The Java projects use Apache Maven[27] for project and dependency management. They are organised as a multi-module project. The top-level `pom.xml` file is an aggregator POM used to build the entire project and to define the versions of common dependencies. The `common` module (sub-project) contains shared code (models, constants), `serialization` contains custom serialisers and deserialisers. `standalone-collectors` contains the collectors based on Confluent Parallel Consumer (CPC). `streams-components` contains the components based on KS – this is, currently, only the data merger.

Collectors based on Confluent Parallel Consumer

Each collector is implemented as a class in the `cz.vut.fit.domainradar.standalone.collectors` package. The project contains two abstract classes:

[26]pandas `apply`: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.apply.html
[27]Apache Maven: https://maven.apache.org/

- `BaseStandaloneCollector` serves as the shared base for collectors. It initialises the Kafka consumers for use with the parallel processor from CPC. It also provides utility methods.
- `IPStandaloneCollector` extends the base with methods specific to the IP-based collectors. It also initialises the single Kafka producer.

The project also contains the `StandaloneCollectorRunner` class, which serves as the entrypoint. It provides a command-line interface, loads the configuration file and initialises the requested collectors. An instance of the Java program may run multiple collectors at once; they are selected using CLI parameters. For example:

```
java -cp [...] cz.vut.fit.domainradar.standalone.StandaloneCollectorRunner
    --col-tls --col-nerd --col-geo -id <prefix> -p ./properties.config
```

executes all three collectors (TLS, NERD, GEO-ASN). The Kafka consumer group ID of each collector will be set to `[prefix]-[collector ID]`. The configuration will be read from `properties.config`.

The data merger based on Kafka Streams

The data merger topology is defined in the `CollectedDataMergerComponent` class. Here, the `StreamsPipelineRunner` class serves as the entrypoint. It provides a command-line interface, loads the configuration file, builds the KS topology and starts the processing. The project is prepared for future extensions, offering the possibility to define multiple KS topologies managed by the runner. The Java program is started with the following command:

```
java -cp [...] cz.vut.fit.domainradar.streams.StreamsPipelineRunner
    --merger -id <KS app ID> -p ./properties.config
```

The runner creates a KS builder, adds the merger topology (`--merger`), and configures the KS application ID, which KS uses internally to construct the consumer group IDs. The configuration will be read from `properties.config`.

Data models and de-/serialisation

Implementations of the data models are in the `cz.vut.fit.domainradar.common.models` package. The models are implemented as plain Java record classes annotated with nullability annotations. Serialisation and deserialisation to/from the JSON format are achieved with the Jackson library[28] and its data-binding component that enables seamless mapping between JSON and simple Java objects. Jackson can be extended with data format modules to provide support for other data formats. The system could easily switch to one of the provided binary formats in the future.

The `cz.vut.fit.domainradar.serialization` package provides implementations of Kafka's `Serializer<T>` and `Deserializer<T>` interfaces used by the strongly typed Kafka consumers and producers. They use Jackson to write and read JSON values. For string values, the Kafka client library already contains an implementation of a (de)serialiser. The package also provides an implementation of `Serde<T>`, a unified factory for a serialiser and deserialiser used in KS.

[28]Jackson library: https://github.com/FasterXML/jackson

Note that when constructing the JSON deserialiser for a generic class, it must be provided with a "type reference object" (note the curly braces):

```
var serializer = new JsonDeserializer<>(jacksonObjectMapper,
    new TypeReference<CommonIPResult<NERDData>>() {});
```

This construct is used to obtain full generics type information by effectively subclassing the `TypeReference` abstract class, as Java normally erases generic types at compile time. This way, an anonymous inheriting class is materialised in the compiled bytecode that carries the type information.

Configuration

The components use static configuration files loaded from the filesystem. Configuration is stored in the line-oriented properties format defined in the `java.util.Properties` Java API[29]. The configuration thus consists of a simple unstructured mapping of string keys to string values. The loaded `Properties` object is passed to the collector instance, as well as to all the Kafka consumers and producers. The properties are defined and documented in the `CollectorConfig` class in the `common` project. An example properties file is included in `client.example.properties`.

All CPC-based collectors offer properties that control the settings of the parallel processor. Notably, the `collectors.parallel.consumer.max.concurrency` controls the number of processing threads spawned by the parallel processor. Other properties are specific for each collector. The configuration file controls the settings of the underlying Kafka consumers and producers.

The KS-based data merger does not define any custom configurable settings. The configuration file may be used to control the behaviour of KS and its consumers or producers.

Note that the CLI provides an optional `-bootstrap-server/-s` parameter that, when used, overrides the `bootstrap.servers` property in the configuration file. In the KS-based components specifically, the command-line parameter `-app-id/-id` is required and always overrides the `application.id` KS property in the configuration file.

Configuration mapping

The configuration manager provides mapping between the properties format used in the configuration files and the JSON format used by the runtime configuration exchange. The JSON strings are handled using the json module from Python; the properties format is simple enough to be read and written using custom code. The configuration exchange object contains two top-level dictionaries (similar to the sections in Python):

- The configuration keys starting with `collectors` are placed in the `collector` dictionary in the exchange object – the `collectors.` prefix is omitted.
- The rest of the keys are placed in the `system` dictionary.

[29]The properties format is specified in the documentation of the `load` method: https://docs.oracle.com/javase/8/docs/api/java/util/Properties.html#load-java.io.Reader-

Note that the manager does not have knowledge of the actual data types in the properties files. When it reads and publishes an existing saved configuration, all values will be strings. In all Java-based collectors, the static configuration (omitted by the manager) includes all properties starting with `ssl` or `security`. An example of a mapping is shown in Listing E.2.

Logging

The components are developed using the SLF4J[30] logging facade, enabling switching of the underlying logging system. The project is configured with the Log4j 2[31] logging framework, which provides rich and flexible logging configuration. The project includes a simple configuration file in `common/resources/log4j2.xml` that outputs logs to the standard output. The `/infra/client_properties/log4j2.xml` file contains an example of a configuration that also sends logs to Kafka using the built-in Kafka appender.

The logger name for the components is constructed as `[full name of component class].[component ID]`. The collectors use the same convention for the use of logging levels as the Python ones. The data merger does not currently log anything.

The general structure of a CPC-based collector

A generalised structure of a collector class is shown in Listing D.3. A collector is a class that:

1. extends `BaseStandaloneCollector<InKey, InValue>`, specifying the types of the key and value of the input topic,
2. calls the superclass constructor with instances of the `Serde` serialiser/deserialiser factory interface for the key and value types,
3. reads its configuration from the provided `Properties` object,
4. initialises a Kafka producer for each output topic,
5. implements the `run` method that builds the parallel processor, subscribes to a topic and starts the processor. The `run` method is called by the runner.

The parallel processor accepts a processing function that is called for each incoming event. The function is executed in a number of threads running in parallel. In the example, as well as in the NERD and TLS collectors, the actual long-running task (remote fetch) is executed in a virtual thread – a Java threading primitive suitable for running tasks that spend most of the time waiting for I/O operations [98]. The processing function calls a custom function that returns a `CompletableFuture` object on which it can wait using `join`. The future object is encapsulated with `orTimeout`, which sets up a monitoring thread that will cancel the future if it does not complete in the specified time.

GEO-ASN collector

The GEO-ASN collector (`GeoAsnCollector.java`) uses MaxMind's GeoIP2 Java library[32] to read data from locally stored GeoLite2 databases. The free-to-use databases are incorporated in this work in accordance with the GeoLite2 EULA[33] and are available from

[30]SLF4J: http://www.slf4j.org/
[31]Log4j 2: https://logging.apache.org/log4j/2.x/
[32]GeoIP2 Java library: https://github.com/maxmind/GeoIP2-java
[33]GeoLite2 EULA: https://www.maxmind.com/en/geolite2/eula

www.maxmind.com. This is the only collector that does not perform any remote fetches, so the I/O operation time is negligible. The processing function is not executed in a virtual thread, and no timeout can be set for it. The collector does not return the `TIMEOUT` code.

NERD collector

The NERD collector (`NERDCollector.java`) uses the bulk querying endpoint[34] from the NERD API, processing incoming data in batches. Batching is built in the parallel consumer and may be configured. The binary format is used to decrease the size of the data sent over the network.

TLS collector

The TLS collector (`TLSCollector.java`) uses Java's built-in SSL/TLS capabilities. It creates a socket, wraps it using `SSLSocketFactory` and performs the handshake. The connect operation and the subsequent socket reads and writes follow the configured timeout. For this reason, the global timeout for processing an event is set to the double of the configured timeout.

Java's SSL/TLS context is initialised with a `TrustManager` responsible for determining if a certificate received from the remote party should be trusted. Because the collector needs to connect to all servers to retrieve the necessary data, a custom dummy implementation, `NaiveTrustManager`, is used to implement the checking method as a no-op, effectively bypassing certificate checking.

The data merger

The data merger uses the Streams DSL to define the stream processing topology. The current implementation closely reflects the structure of the operation defined in Listing B.2. It aggregates all IP data per a domain name using the `groupByKey` and `groupBy` operations. The conceptual `COLLECT_TO_MAP` operation is realised with the `aggregate` operation. All result streams are interpreted as tables (leveraging the duality of streams and tables) that are ultimately joined to create the `AllCollectedData` object. The two filtering operations are implemented using the `hasEnoughIpCollectorResults` and `hasTlsIfRequired` methods. The expected IP results filtering is slightly weakened to also pass results that do not yet have NERD results.

The implementation simplifies the design from Section 6.5. When using a stateful operation or representing a stream as a table, the framework maintains a *state store*. It is similar to the "most useful state store" but uses offset-based semantics: the entries are only compared by the event's offset in the Kafka log. The topology is defined so that the stream of results of the DN-based collectors is directly used as a table changelog. Thus, this table does not follow the usefulness ordering – the last produced entry will always be present in the resulting merged data object. However, this should not affect the operation of the system, as no collector should produce more than one result unless specifically requested (*e.g.* by the re-collection controller that could account for the required ordering). Conversely, the results from the IP-based will follow the usefulness ordering as the stream is not converted to a table directly. Instead, a table is created using the `aggregate` operation with a custom aggregator function that can easily implement the required logic.

[34]NERD bulk querying API: https://github.com/CESNET/NERD/wiki/API#bulk-querying

Additionally, the component currently does not evict entries from the state stores backing the materialised tables. This will eventually become a problem, as it allows the stores to grow indefinitely. It seems that this could be solved using the windowing capabilities of KS: "session windows" are currently used for the first IP data aggregation operation that limits the maximum time between two records related to the same DN/IP pair. However, the extension of this mechanism to the DN-based collectors is a subject of ongoing experiments and has not yet been included in the implementation.

7.6 Database Integration using Kafka Connect

Kafka Connect is used together with the JDBC Source and Sink connectors by Confluent[35], the MongoDB Connector[36] and several custom converters and transformations located in the `/src/java_pipeline/connect` directory. The section will also refer to configuration files located in `/infra/connect_properties`. PostgreSQL stores the input domain names and the metadata on the last collection attempts. MongoDB stores the entire collected data objects, and the feature vectors (if configured).

Domain names input

The table for the input domain names is defined by the snippet of SQL in Listing 7.1a. Note that it uses the `serial` data type that creates an automatically incrementing sequence for the IDs and creates a unique index on the domain name column. The `filter_output` attribute is only populated when a domain name is filtered from processing but should be transferred to MongoDB for display in the UI.

The loader uses the SQL in Listing 7.1b to perform the upsert operation for any number of filtered input domain names: The unique index and the `ON CONFLICT` clause ensure that when the list of values contains a DN that has already been added, the existing entry is only updated with the current timestamp.

The JDBC Source connector configuration in `40_postgres-source.properties` ingests the domain names from the table and produces them to the *to_process_zone* topic. The incremental mode of the connector works by periodically polling the database for new records, internally keeping track of the last ingested record's ID. It is configured with the query shown in Listing 7.1c to select the domain names. The query is encapsulated in an extra `SELECT` because the connector appends a `WHERE` clause to it, comparing the value of `id` to the last seen value.

The connector converts database entities to internal key-value message representations called records. Each is associated with an abstract data type for both the key and the value, specified using a `Schema`[37]. Records created by the JDBC connector have empty keys, and values are constructed as structures with fields inferred from the database columns. The target topic expects a domain name as a plain string in the key, so a KC transformation must be used that extracts the `domain` field from the value and sets it as the key. This custom transformation is implemented in the `PostgresIngressTransformation` class.

[35]Confluent JDBC connectors: `https://docs.confluent.io/kafka-connectors/jdbc/current/`
[36]MongoDB Connector: `https://www.mongodb.com/docs/kafka-connector/current/`
[37]`Schema`: `https://kafka.apache.org/37/javadoc/org/apache/kafka/connect/data/Schema`

```sql
CREATE TABLE domains_input (
id       serial
         PRIMARY KEY,
domain   text
         NOT NULL UNIQUE,
first_seen  timestamptz
            NOT NULL DEFAULT now(),
added    timestamptz
         NOT NULL,
filter_output jsonb);
```

(a) DDL for the input domain names table

```sql
INSERT INTO domains_input
  (domain, last_seen, filter_output)
VALUES
  ('domain name #1', now(), NULL),
  ('domain name #2', now(), NULL),
  ...
ON CONFLICT (domain)
DO UPDATE SET
  last_seen = now()
RETURNING *;
```

(b) the loader's upsert SQL

```sql
SELECT * FROM (
  SELECT id, domain
  FROM domains_input
  WHERE filter_output IS NULL
)
```

(c) the query configured for the connector

```sql
SELECT * FROM (
  SELECT id, domain, filter_output
  FROM domains_input
  WHERE filter_output IS NOT NULL
)
```

(d) the query configured for the filtered domains passthrough connector

Listing 7.1: The SQL queries related to the input domain name table.

PostgreSQL collection metadata output

As a proof of concept, the current configuration also stores the metadata on the last collection attempts in the PostgreSQL database. They are not used in the current implementation but may serve as the input for a re-collection controller or for quick diagnostics. The tables for the DN-based and the IP-based collectors are defined in Listing 7.2. Note that the `inet` type could be used to store the IP address, but the connector does not support it.

The main output topics of all DN-based collectors (*processed_*) are consumed by the JDBC Sink connector configured in `30_postgres-sink-dn-collectors.properties`. Sink connectors must be configured with implementations of `Converter`[38] that translate between KC records and the binary data stored in Kafka. The built-in string converter is used for keys, while values use a custom converter, `CommonResultConverter`, that deserialises the JSON value and creates a KC structure that matches the scheme of the target tables. The converter also infers the collector ID from the topic name, so a single connector instance can be used for all (DN-based) collectors.

The connector performs the upsert operation to ensure that the table always contains at most one record for each pair of a domain name and a collector. In order to use the upsert mode, the collector needs to be configured with the key fields, and these must be found within the KC record's key. This is ensured by a custom transformation, `CollectorInValueToKeyTransformation`, that creates the appropriate key structure from the fields in the value.

[38]`Converter`: https://kafka.apache.org/37/javadoc/org/apache/kafka/connect/storage/Converter

```sql
CREATE TABLE dn_collectors_states
(
  domain_name  text      NOT NULL,
  collector    text      NOT NULL,

  last_attempt timestamp NOT NULL,
  status_code  smallint  NOT NULL,
  error        text,
  CONSTRAINT dncol_pk PRIMARY KEY
    (domain_name, collector)
);
CREATE INDEX dn_states_index ON
  dn_collectors_states (domain_name);
```

(a) DN-based collectors metadata table

```sql
CREATE TABLE ip_collectors_states
(
  domain_name  text      NOT NULL,
  ip           text      NOT NULL,
  collector    text      NOT NULL,
  last_attempt timestamp NOT NULL,
  status_code  smallint  NOT NULL,
  error        text,
  CONSTRAINT ipcol_pk PRIMARY KEY
    (domain_name, ip, collector)
);
CREATE INDEX ip_states_index ON
  ip_collectors_states (domain_name);
```

(b) IP-based collectors metadata table

Listing 7.2: The DDL scripts for the collector results metadata tables.

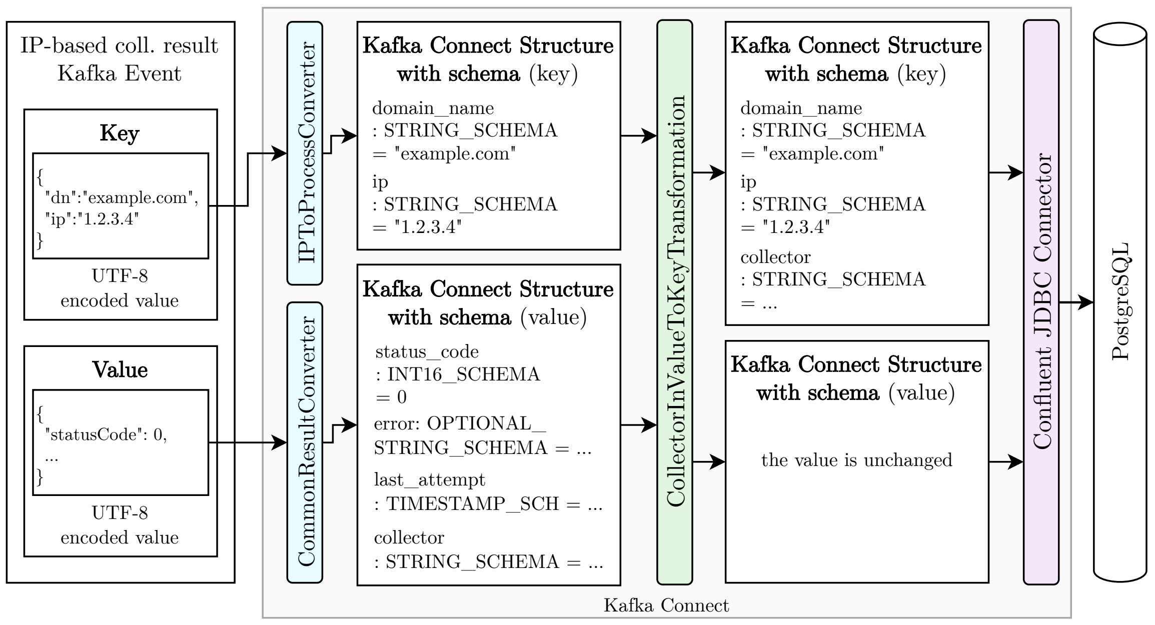
The IP-based collectors are handled by a similar but separate connector instance configured in `31_postgres-sink-ip-collectors.properties`. The custom value converter and the transformation are the same, as they were designed to handle both cases. Events in the *collected_IP_data* topic use complex values as keys, so another custom converter, `IPToProcessConverter`, is used. The process is demonstrated in Figure 7.6.

MongoDB collection data output

MongoDB serves as a persistent storage for all collection results (and, in DomainRadar, for classification results, too). Results from all DN-based collectors are put in a single collection, `dn_data`, and results from IP-based collectors are put in `ip_data`. The key (`_id` field) of each MongoDB document is composed of a domain name, a collector ID and a timestamp, unambiguously identifying a result (results from IP-based collectors also include the IP address). Listing 7.3 shows an example of how a collector result object maps to the stored MongoDB document. Observe that for debugging and analysis purposes, the document also includes the Kafka event offset and partition.

The MongoDB sink connector for all DN-based collectors is configured in `20_mongo-sink-dn-collectors.properties`. The built-in string converter is used for keys; values are converted using the built-in JSON converter. A custom transformation, `MongoKeyTransformation`, creates the KC structure that will be mapped to the `_id` field. The built-in `InsertField` transformation from KC inserts the offset and partition fields to the value.

The MongoDB connector also provides post-processing chains that modify the BSON document just before it is inserted into the database. The built-in `DocumentIdAdder` processor with the `FullKeyStrategy` creates the `_id` field from the record's key. The `BlockListValueProjector` removes the `lastAttempt` field from the document, as it was previously copied to the key. Finally, the `ReplaceOneDefaultStrategy` write strategy is used to replace the document if the same key already exists. This can happen due to the at-least-once nature of the message delivery. For the IP-based collectors, the configuration is nearly the same, except that the keys are again converted using `IPToProcessConverter`.



IP-based coll. result Kafka Event
Key
{ "dn":"example.com", "ip":"1.2.3.4" }
UTF-8 encoded value
Value
{ "statusCode": 0, ... }
UTF-8 encoded value
IPToProcessConverter
CommonResultConverter
Kafka Connect Structure with schema (key)
domain_name : STRING_SCHEMA = "example.com"
ip : STRING_SCHEMA = "1.2.3.4"
Kafka Connect Structure with schema (value)
status_code : INT16_SCHEMA = 0
error: OPTIONAL_ STRING_SCHEMA = ...
last_attempt : TIMESTAMP_SCH = ...
collector : STRING_SCHEMA = ...
CollectorInValueToKeyTransformation
Kafka Connect Structure with schema (key)
domain_name : STRING_SCHEMA = "example.com"
ip : STRING_SCHEMA = "1.2.3.4"
collector : STRING_SCHEMA = ...
Kafka Connect Structure with schema (value)
the value is unchanged
Confluent JDBC Connector
PostgreSQL
Kafka Connect

Figure 7.6: The process of transferring a Kafka event with a result from an IP-based collector to the PostgreSQL database. The blue blocks represent the custom converters, the green block represents the custom transformation, and the pink block represents the JDBC Sink connector.

Filtered domains passthrough

The filtered domains, where `filter_output` is not null, are ingested to Kafka by a separate PostgreSQL connector instance configured in `41_postgres-source-filtered.json` only to be consumed by a MongoDB sink connector configured in `21_mongo-sink-filtered-domains.properties`. The source connector uses the query in Listing 7.1d to select the records and transforms them using built-in transformations so that the resulting key is the domain name, and the value is the JSON from `filter_output`. The sink connector transforms the incoming records so that the key is composed of the domain name and the Kafka event's timestamp; the rest of the document corresponds to the value in the record.

Feature vectors

When the JSON output of the feature extractor is enabled, the MongoDB sink configured in `24_mongo-sink-feature-vectors.properties` stores the feature vectors in the `feature_vectors` collection. The configuration is essentially the same as for the filtered domains.

MongoDB aggregations

The `/infra/mongo_aggregations` directory contains several examples of MongoDB aggregations that process the stored collection results data. The `results.js` file contains the aggregation used for the DomainRadar UI. The aggregation in `all_raw_data.js` reconstructs the merged data objects. The files can be executed using the `mongosh` MongoDB shell:

```
mongosh "mongodb://[mongo URI]" --file [aggregation_file.js]
```

```
{



  "lastAttempt": 1719813730907,

  "statusCode": 0,
  "error": null,


  "rdapTarget": "my.zone.com",
  "rdapData": {
    "...": "[RDAP response fields]"
  },
  "...": "[other fields]"
}
```

(a) a collector result object

```
{
  "_id": {
    "domainName": "input.dn.zone.cz",
    "collector": "rdap",
    "timestamp": 1719813730907
  },
  "statusCode": 0,
  "error": null,
  "k_offset": 1234,
  "k_partition": 12,
  "rdapTarget": "zone.cz",
  "rdapData": {
    "...": "[RDAP response fields]"
  },
  "...": "[other fields]"
}
```

(b) the corresponding MongoDB document

Listing 7.3: An example of the storage format for the collector results.

At the end, all the files contain a call to a custom utility function defined in `common.js`. When executed, it materialises the results of the aggregation pipeline into a collection.

7.7 Container Images

To simplify the testing and deployment, Dockerfiles[39] are provided to build container images for all the components. They have been tested and shown working using Docker (26.1.3) with buildx (0.14.1), as well as podman (5.0.2) with buildah (1.35.3).

The bash script in `/src/build_images.sh` is provided to build the images for all pipeline components at once. The images are tagged as `domrad/[tag]` where `[tag]` is the collector ID (for Python-based components), `java-streams`, `java-standalone` (for Java-based components), `kafka-connect` (for the KC image), `standalone-input` or `config-manager`. For more information on the script, execute it with the `-h` (help) option. For examples of the use of the individual images, refer to the `/src/README.md` file and the included Compose file.

Images for Python-based components

The shared Dockerfile for the Python-based collectors and the feature extractor is in `/src/python_pipeline/Dockerfile`. The build is controlled by the `TARGET_UNIT` argument that chooses the target project directory and the `TARGET_MODULE` argument that chooses the Python module name.

The resulting image is based on the official Python 3.11 (on Debian 12) image. The Dockerfile[40] installs Poetry, the system packages needed for building dependencies, and the dependencies. Cache mounts are used so that when the build is repeated, the dependencies

[39]I am keener on the "Containerfile" term, but for technical reasons, the project was mostly developed with Docker, hence this naming of the file.

are cached on the host system. The final runtime image does *not* include Poetry and the build packages; only the dependencies and project sources are copied from the build stage. The entrypoint is set to the `docker_entrypoint.sh` script the executes the Faust application from the specified Python module.[41]

Images for Java-based components

The Dockerfiles for the Java-based components are located in `/src/java_pipeline`. The `standalone.Dockerfile` one builds the image for the collectors. `streams.Dockerfile` builds the image for the data merger.

The build stage is based on the Eclipse Temurin 21 distribution of OpenJDK[42] (on Ubuntu 22.04); it installs Maven and uses it to build the project, producing a single JAR file. The final runtime image is based on the Temurin Java Runtime Environment (JRE) image. Only the JAR file is copied to it, keeping the image size small. The image for Kafka Streams additionally includes the jemalloc allocator library to improve the performance of the Kafka Streams application.

Kafka Connect image

The `/src/java_pipeline/connect.Dockerfile` file contains two stages: `build` sets up Maven, builds the `connect` Java project with the custom KC plugins, and exports the built JAR file together with the dependencies in standalone JAR files. The `runtime` stage prepares a standalone KC image based on Temurin JRE: it downloads the Kafka 3.7.0 distribution, decompresses it and copies the JAR files from the previous stage to a plugins directory. This image is then used by the Compose file (see below).

Other images

`/src/config_manager/Dockerfile` and `/src/standalone_input/Dockerfile` are Dockerfiles that build images for the configuration manager and the standalone input, respectively. The Dockerfiles are essentially the same as for the Python-based pipeline components, except that they specify another entrypoint and offer no build arguments.

7.8 Compose-based Orchestration

The system is composed of ca. 20 microservices, including Kafka and the database systems. The attached storage contains the `/infra` directory with a Compose file[43] and a set of configuration files and scripts that can be used to start the system in a local environment easily. The Compose file was tested with Docker Compose (2.27.1). For production and large-scale use, a more sophisticated container orchestration system (such as Kubernetes) should be used.

The Compose file defines the following services:

- `kafka1` is a single-node Kafka broker,

[41]The Dockerfile was partially adopted from a publicly available Gist: https://gist.github.com/usr-ein/c42d98abca3cb4632ab0c2c6aff8c88a, cit 2024-07-01.
[42]Eclipse Temurin: https://adoptium.net/en-GB/temurin/releases/
[43]See the Compose specification website for more information: https://compose-spec.io/

- `kafka-connect-full` is a standalone KC instance with the full configuration, *i.e.* all the connectors are enabled,
- `kafka-connect-without-postgres` is a standalone KC instance used in the standalone collector scenarios, configured not to include the PostgreSQL connectors,
- `kafka-ui` (Kafbat UI[44]) is a web-based user interface for Kafka management, included for ease of testing,
- `initializer` waits for Kafka to start and executes the topic initialisation script; it defers the execution of the other services until Kafka is ready,
- `collector-[collector ID]`, `merger`, `extractor`, `config-manager`, `standalone-input`, `postgres`, and `mongo` are self-explanatory,
- `mongo-{domains, raw-data}-refresher` periodically refresh the materialised views in the MongoDB database.

The single Compose file can be used in several profiles for different use cases:

- The `full` profile starts all the services except `kafka-connect-without-postgres` and `standalone-input`. This configuration simulates the DomainRadar system with all the components (although the loader, the classifier unit, and the UI are not included).
- The `col` profile starts Kafka, Kafbat UI, MongoDB, the PostgreSQL-less KC service and all the collectors. This configuration is used for the “standalone collector only” scenario.
- The `colext` profile extends the previous one with the data merger and the feature extractor. This configuration is used for the “standalone collector and feature extractor” scenario.

For example, the full configuration is started in the background using:

```
docker compose --profile full up -d
```

Additionally, the scaling of the services (collectors, extractor, merger) can be controlled using the variables set in the `.env` file.

Exposed services

The Compose file exposes the following services to the host system (all ports are TCP):

- the Kafka broker on port 31013,
- the Kafbat UI web application on port 31000,
- the KC HTTP API on port 31002,
- the PostgreSQL database on port 31010,
- the MongoDB database on port 31011.

[44]Kafbat UI: https://github.com/kafbat/kafka-ui

To enable easier firewall management on the host system, if required, the services actually communicate with each other using isolated networks. The ports are only exposed to the host system when they are added to non-isolated networks. The Compose configuration includes two such networks: `kafka-outside-world` used only for Kafka, and `outside-world` used by the other services. All services are assigned static IPs in these networks. This way, it would be possible to remove Docker's port forwarding and instead set up static routing rules that forward traffic to the containers.

Security setup

Kafka is configured to use SSL/TLS for authentication and encryption. Before the first start, SSL/TLS certificates must be generated for the broker and all the clients. The bash script in `/infra/generate_secrets.sh` creates a custom certification authority (CA), generates a required number of RSA keypairs and CSRs for the brokers and the clients, and signs them with the CA. The generated secrets are saved in the `secrets` directory; their use is pre-configured in the clients' configurations. The generator script requires OpenSSL and JRE. Alternatively, the `/infra/generate_ secrets_docker.sh` script is provided to generate the secrets in a container.

The passwords for the exported keys and the keystores, the number of client and broker certificates to generate, and their lifetime can all be set through the variables at the beginning of the script. If deploying in production, the passwords should be changed, and the CA private key must be stored securely. The `generate_new_client_secret.sh` script can be used to generate additional client certificates (*e.g.* for future extensions of the system).

Moreover, the `/infra/db` directory contains the database initialisation scripts that create database objects and users with passwords specified in the `.secret` files therein. For PostgreSQL, the created users are `master` (a superuser), `connect` (for KC), `controller` (for a re-collection controller) and `prefilter` (for the loader). Only the `connect` user is used in the current implementation; the rest serve as placeholders for future development. For MongoDB, the created users are `master` (a superuser) and `connect` (for KC). The initialisation scripts are executed by the database services automatically. Again, the passwords should be changed before deployment.

Using the configuration manager

When the manager applies a new configuration, it must restart the affected service. To do this, the manager invokes the `docker compose` command. However, when used in a container, the manager cannot access the host's Docker Compose directly. For this reason, a remote mode is included. The `/src/config_manager/config_manager_daemon.py` script must be first started on the hosting machine. It creates a Unix socket that is then mounted to the manager container. The script listens for commands from the manager and executes them using the Docker Compose CLI. It only requires pure Python (tested on version 3.8).

The `config-manager` service is not included in any of the aforementioned Compose profiles. To use the manager, start the daemon and add the `configmanager` profile:

```
# Start the daemon with the correct Compose command
python config_manager_daemon.py "/tmp/domrad_control.sock" \
    "docker compose --profile full -f /path/to/compose.yml" &
# Start the config-manager service
docker compose --profile full --profile configmanager up -d
```

Note that the configuration manager is rather experimental and mainly included as a proof of concept.

Using the standalone input controller

To use the system as a standalone collector, start the `col` or `colext` profile in the background. The `standalone-input` service can then be executed using `docker compose run` to load domain names for processing or to start re-collection. For example, to load domain names from a local file, use the following commands:

```
# Start the system
docker compose --profile colext up -d
# Run the loader
# Remember to mount the file to the container
# -d for direct mode, i.e. one domain per line
docker compose --profile colext run \
    -v ./my/source/file.txt:/app/source.txt \
    standalone-input load -d /app/source.txt
```

The progress can be monitored using Kafbat UI or by inspecting the MongoDB collections.

Chapter 8

Performance and Reliability Evaluation

The implemented system was tested in both main use cases: for standalone collection and processing of a large input batch and for real-time data collection for classification in the DomainRadar system. The experiments targeted determining the system's behaviour and performance in different scaling, threading, and concurrency configurations. Additionally, the composition of erroneous responses was analysed to identify potential bottlenecks and to evaluate the system's reliability.

The system was executed on two virtual machines (VM) in the CESNET network, both running in the VMWare ESX 6.7.0 hypervisor on a server based on the Intel Xeon Gold 6226R CPU at 2.9 GHz. The *infra* VM had 6 virtual CPUs, 48 GiB of memory, and 500 GiB of SSD storage; it ran Debian 12.6 on Linux kernel version 6.1.0. It was used to run all the services except for the collectors (Kafka, Kafka Connect, the database systems, the data merger and the feature extractor, the DomainRadar loader and classifiers). The *scanner* VM had 4 virtual CPUs, 16 GiB of memory, and 100 GiB of SSD storage; it ran Oracle Linux Server 8.10 on Linux kernel version 5.4.17. It was used to run the collectors. The connection speed between the two VMs was measured using iperf3[1] to be about 3.5 Gbps. This high-speed connection ensures minimal latency and efficient data transfer between the components, supporting the system's performance under high load.

The VM setup was chosen to isolate internet-facing services from strictly internal ones, thereby mitigating risks of external attacks targeted at the entire system. *Infra*, running the infrastructure, was placed inside a "private network". It never opened internet connections, it did not have a publicly routable IP address and could only be accessed through a VPN. *Scanner*, used to fetch data from the internet, was placed in a "public network". It had a public IPv4 address and served a static webpage with a notice on the scanner's purpose. It could only reach the private network via the TCP port of Kafka running on *infra*.

This chapter provides an overview of the experiments performed to evaluate the system's performance and reliability. The findings emphasise its ability to handle high throughput and the identification of possible bottlenecks. The experiments are documented with tables and graphs showing the system resource usage, plots showing how the number of responses from collectors rised over time, and tables and charts breaking down the composition of errors in the responses from the collectors. They are available in Appendix H.

[1]iperf3: `https://iperf.fr/`

The forms of parallelism

The implemented system offers two approaches to parallelism in the pipeline components. First, all the components may be *scaled horizontally* by increasing the number of partitions in Kafka and running more independent instances of the collector process. In practice, the instances could be executed on multiple machines, scaling the system *out*. Kafka's consumer group mechanism splits the work across the running instances. While starting the component in more instances than the number of partitions is possible, they would idle unless an active instance crashes. This chapter uses the following symbols to denote the number of partitions and running instances:

- P_C refers to the number of partitions set for the *to_process_[...]* Kafka topics used for the collection requests, setting the maximum number of collector instances;
- P_M refers to the number of partitions set for the *processed_[...]* and *collected_IP_data* topics, setting the maximum number of instances (or processing threads) of the data merger;
- P_E refers to the number of partitions set for the *all_collected_data* topic, *i.e.* the maximum number of instances of the feature extractor;
- C_{Py}, C_J, M, E refer to the number of running individual instances of the Python-based collectors, Java-based collectors, the data merger, and the feature extractor, respectively.

Second, the components offer fine-grained control over the parallelism of the processing tasks inside each instance:

- The Python-based collectors have the concurrency setting C_{Py}^c, which affects the number of independent tasks planned in the event loop.
- The Java-based collectors offer to control the number of threads C_J^t in which the Parallel Consumer library processes the incoming requests. (Observe that this setting does not depend on the number of partitions.)
- Kafka Streams, used in the data merger, splits the input topic-partitions into tasks. A single instance of the merger can create M^t threads in which the tasks are processed. Each thread gets an independent consumer. As a result, the processing capacity should be similar if $M = 1$, $M^t = N$ and if $M = N$, $M^t = 1$, although the system resource usage may differ. (Observe that this setting depends on the number of partitions: to increase the effective processing capacity, it must hold that $N \leq P_M$.)
- Each instance of the feature extractor may spawn E^t extra processes to which the requests (accepted by the main process of the instance) are distributed. When changing E^t, the concurrency setting is also changed to the same value so that Faust can use the extra processes in parallel.

Additionally, the feature extractor has a setting E^b that controls the batch size. In the current implementation, it should not affect the amount of parallelism much. However, it can affect on the resource usage and processing time.

8.1 Standalone Collection Experiments

The goal of the standalone collection experiments was to determine the appropriate levels of horizontal scaling for each collector, as well as the parallelism and concurrency configuration inside of the instances.

This section first describes the individual experiments from the performance perspective. Then, it discusses the observed collection errors and the expected load on the Kafka broker. The results of these experiments are available in Appendix Section H.1.

Method

The steps repeated for each experiment were:

1. Stop the running system, back up and clear the data directories (Kafka, MongoDB).
2. Adjust the partitioning configuration in `prepare_topics.sh`.
3. Adjust the collector scaling options in `.env` and the collector configuration files.
4. Start the infrastructure (Kafka, MongoDB, KC) and wait for initialisation.
5. Start the collectors and wait for the completion of consumer rebalancing.
6. Load the testing domain list at once (to the *to_process_zone* topic).
7. Wait for all the consumers to reach zero lag (*i.e.* all requests have been consumed).

The collection results were transferred to a MongoDB database (through KC), where they were later retrieved and analysed using the scripts in `/analysis`. The data merger and the feature extractor were not used. Data on the system resource usage were collected via the collectd[2] monitoring tool.

All experiments used the configuration from `/infra/client_properties`, differing only in the concurrency and threading settings mentioned above. The only exception was that compression was turned off at the producers (in the attached files, it is set to ZSTD). The local RDAP rate limiters were kept at the default setting: the queued mode with no time bound, a maximum rate of 5 requests per second, and a maximum burst of 5 requests.

Observed performance metrics

The following performance metrics were observed for each collector:

- *Total collection time* (Tot, in hours and minutes) is the time elapsed between the first and the last response of the collector.
- *Average throughput* (AvgTput, in processed requests per second) is the total number of responses produced by the collector, divided by its total collection time. This metric expresses the expectation of how many domain names (or IPs) the collector can process "on its own".
- *Average collection time* (AvgColT, in milliseconds per request) is the inverse of AvgTput, expressing the expectation of time needed to process a single domain name (or IP).

[2]collectd: https://www.collectd.org/

- *Mean queued/collection time* (MnQdColT, in seconds per request) calculates the average time between a collection request and its corresponding response. This metric provides an estimate of the latency in the collector and reflects how well the collector performed under a particular workload, taking into account requests that were waiting in a queue at its input. Generally, this value should increase when the collector is handling more requests and decrease when the throughput of requests increases (for example, because of higher parallelism). The request timestamp is approximated with the time of the response from the previous stage.

Each experiment is accompanied by two charts showing how each collector's total number of responses evolves over time, separately for the DN-based and the IP-based collectors. They visualise how much the collectors "lag behind" their input stages, *i.e.* the closer the lines are to each other, the better.

Experiments #3 and #4 also include details on the system resource usage. "CPU usage" refers to the relative CPU time spent in userspace, averaged across all cores. The graphs show per-core relative CPU time spent in userspace and an across-cores average CPU time spent in kernel and in userspace. The system load is a value reported by the Linux kernel that shows the number of tasks waiting in the run queue (each sample measures the average over 5 minutes). The network interface usage shows the traffic on the interface that connects the machine to the internet. The memory usage is calculated as `MemTotal - MemFree - Buffers - Cached - SReclaimable` (see the `proc_meminfo(5)` Linux manual page for more information).

Input data

The testing list of 400,000 domain names was created from the CESNET data used in Section 5.2. First, all domains belonging to one of the top 15 resolved eSLDs or one of the top 15 resolved e3LDs (sorted by subdomain count, see Table A.2) were removed. The domains were then randomly sampled to make the list (available in `/analysis/testing_domains.txt`). This particular number was chosen to mimic the expected throughput of the system in a real-world scenario: The assumption was that if the collectors manage to process the list in a couple of hours, the system should not get congested when processing the live feed where a similar amount of domain names will be spread out over the day.

The failed experiment

After initially setting the system up in the testing environment, it was executed with $P_C = C_{Py} = C_J = 16$, mainly to verify the overall function in a high-load scenario. This configuration seemed too demaning for the *scanner* VM, where it used up almost all memory and CPU resources. However, the critical observation was that the Python-based collectors were randomly starting to hang: the instances were running but not consuming any messages or producing output. This misbehaviour may have been related to issues 175[3] and 523[4] tracked in the Faust repository that reported Faust agents dying over time.

The issue was mitigated by increasing the timeout settings for various low-level Kafka operations and decreasing the maximum polled records limit. The collectors always successfully processed all the data with these adjustments in place. They were incorporated into the Compose file. However, the charts in Figures H.1 and H.7 show the zone collector

[3]Faust issue #175: https://github.com/faust-streaming/faust/issues/175, cit. 2024-07-11.
[4]Faust issue #523: https://github.com/faust-streaming/faust/issues/523, cit. 2024-07-11.

sometimes stopping randomly for a few minutes. The reason for this behaviour is unknown but may be related to the same Faust issue. When running in real time, the issue was not observed.

Experiment #1: 2 partitions

The first experiment was set up with $P_C = C_{Py} = C_J = 2$ and $C^c_{Py} = C^t_J = 4$. The configuration had been chosen to start with a low amount of parallelism and get some baseline results in a reasonable time. Table H.1 shows the per-collector metrics.

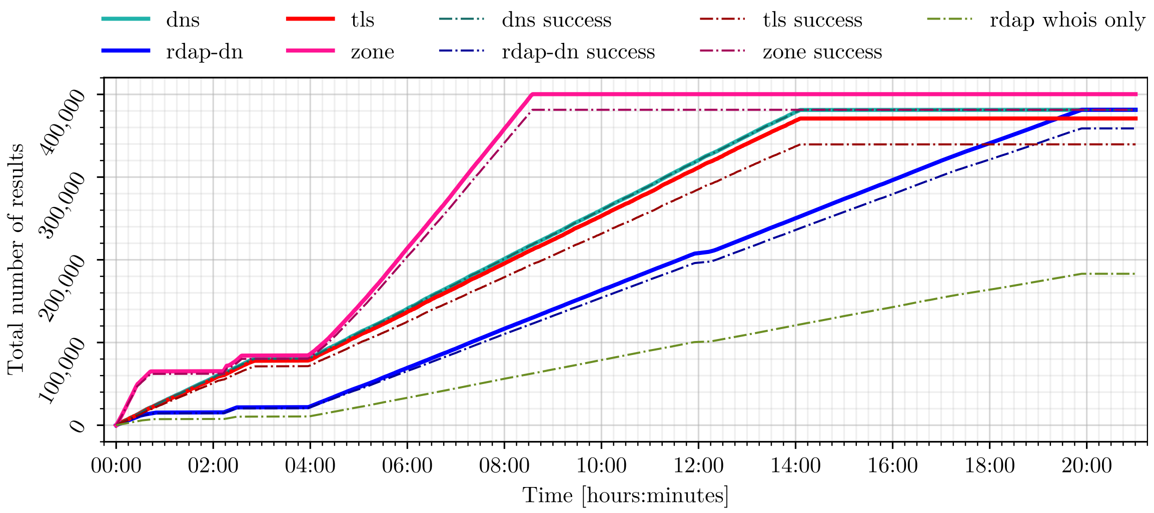


Figure 8.1: Number of responses from the DN-based collectors over time in experiment #1. Random periods of inactivity and the struggle of the DNS and RDAP collectors are visible. In contrast, the TLS collector line closely follows the DNS one, showing low processing latency.

Findings:

- The zone collector processed all entries in 8.5 hours, with the dependent DNS collector lagging considerably, finishing in 14 hours (see Figure 8.1). This was expected as the DNS collector performs more queries per entry. It suggests the need for more instances of the DNS collector.
- The Java-based TLS, GEO-ASN, and NERD collectors were clearly underutilised and collected the IPs coming from the DNS collector almost immediately, apparent from the low values of MnQdColT. They may not require higher scaling.
- Both RDAP collectors processed the entries in over 19 hours, lagging behind the DNS collector. High values of MnQdColT suggest that domain names long before being processed, likely due to the rate limiter in the RDAP collectors slowing the processing down. In addition to scaling, the optimal rate limiter settings should be further evaluated.
- The RTT collector was extremely slow: after 21 hours, it had only processed under 10,000 domain names. The experiment was terminated here.

- The usage of the system resources was not metered in this experiment due to technical issues. However, manual observation showed that the system was underutilised, with the CPU usage not exceeding 15% and the memory usage not exceeding 3 GiB.

Experiment #2: 4 partitions

In the second experiment, the partitioning was doubled to $P_C = 4$. The Python-based collectors were also set to $C_{Py} = 4$ instances, as they were the bottleneck in the previous experiment. The Java-based collectors were kept at $C_J = 2$ instances because they handled the throughput well in the previous experiment. The concurrency and parallelism were kept at $C^c_{Py} = C^t_J = 4$ to observe the effects of scaling only. Table H.3 shows the resulting per-collector metrics.

Findings:

- Most collectors showed over 50% decrease in total processing time compared to the previous experiment. Excluding the RTTs, the batch was processed in below 10 hours – the zones were processed in 5:43, the DNS requests took 6:34.
- The QdMnColT values of the Java-based TLS, GEO-ASN and NERD collectors increased, showing how the collectors were more utilised. The increase was especially significant in the TLS collector, showing the need for higher parallelism if the throughput grows.
- The QdMnColT value decreased for the DNS collector, showing that the additional parallelism helped the collector to process faster even though the load was higher (as the zone collector was also faster).
- QdMnColT also decreased for the RDAP collectors. This is likely because the instances do not share the rate limiter, allowing them to contact the RDAP servers more often overall. This is also supported by the error breakdown charts showing a slight increase in both the total amount of errors and the `RATE_LIMITED` results. However, both RDAP collectors still lagged behind the others at 8:23 (IP) and 9:56 (DN).
- The RTT collector still struggled with the throughput, processing only about 150,000 requests in 22 hours.
- The usage of the system resources was still small. The CPU usage did not exceed 30%; the peak memory usage was about 4.5 GiB.

As the RTT collector was struggling, it was terminated after about 23 hours and reconfigured with $C^c_{Py} = 64$. After this change, the throughput increased considerably (see Figure 8.2), showing that the concurrency setting affects the process positively.

Experiment #3: 8 partitions

In the third experiment, the partitioning was set to $P_C = 8$. The Python-based collectors were set to match the partitioning $C_{Py} = 8$; all the collectors except for the RTT one were set to $C^c_{Py} = 4$. The RTT collector was set to $C^c_{Py} = 64$ based on the findings from the previous experiment. The Java-based collectors were kept at two instances ($C_J = 2$), but

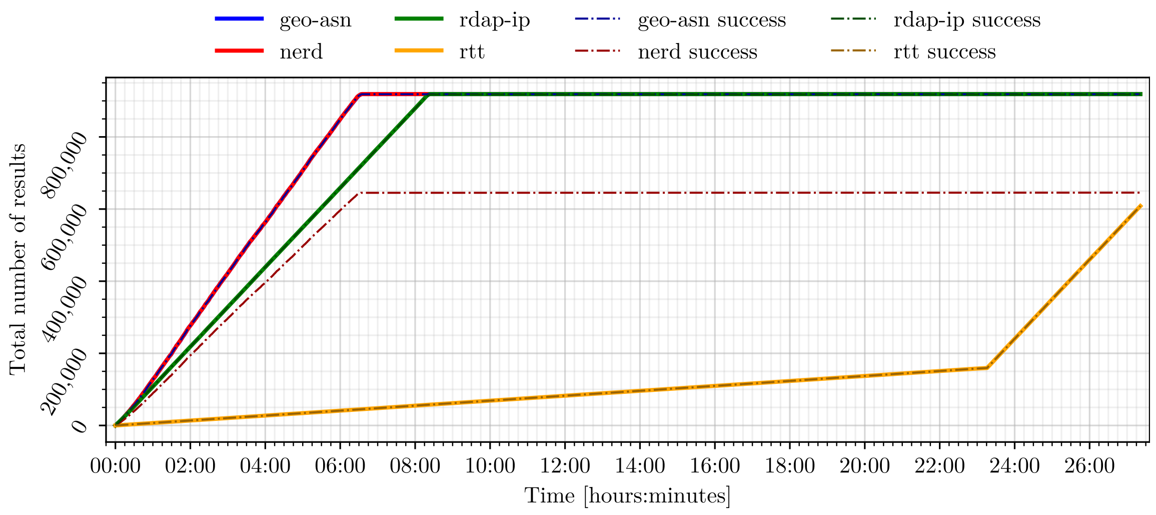


Figure 8.2: Number of responses from the IP-based collectors over time in experiment #2. Note how the RTT lags behind but increases the throughput significantly after the reconfiguration.

the number of threads was increased to $C_J^t = 8$ to account for the additional input load. Table H.5 shows the per-collector metrics. Figure H.5 shows the response accumulation and the system resource usage measurements.

Findings:

- The average CPU usage was 31%; the peak was 39%. The average memory usage was 7.91 GiB; the peak was 9.54 GiB.
- The performance of the zone collector was almost the same as in the previous test (5:29 vs 5:43), which is somewhat surprising, given the improvements in the similarly operating DNS collector.
- On the contrary, the performance of the other Python-based collectors improved significantly. As shown in Figure 8.3, the DNS, RDAP-IP, and RTT collectors did not lag significantly behind their input; their total processing time (5:29) was also equal to the one of the zone collector. In the case of the RTT collector, the MnQdColT value shows that it was likely approaching its limits.
- The RDAP-DN collector took only 29 minutes more than the other collectors, significantly improving compared to the previous experiment. Its overall error rate increased by 2.4%.
- The Java-based collectors had no issues with the throughput.

The results show the system can handle real-time traffic of approximately 20 domain names per second in this configuration. This conclusion is based on the observed throughput of the zone collector and the fact that the other collectors did not lag behind it (although the RDAP-DN rate limiter configuration must be tuned). However, the memory usage chart (see Figure H.5) shows an alarming trend: memory usage was steadily increasing over time, suggesting a memory leak in one of the components. This is a subject for further evaluation.

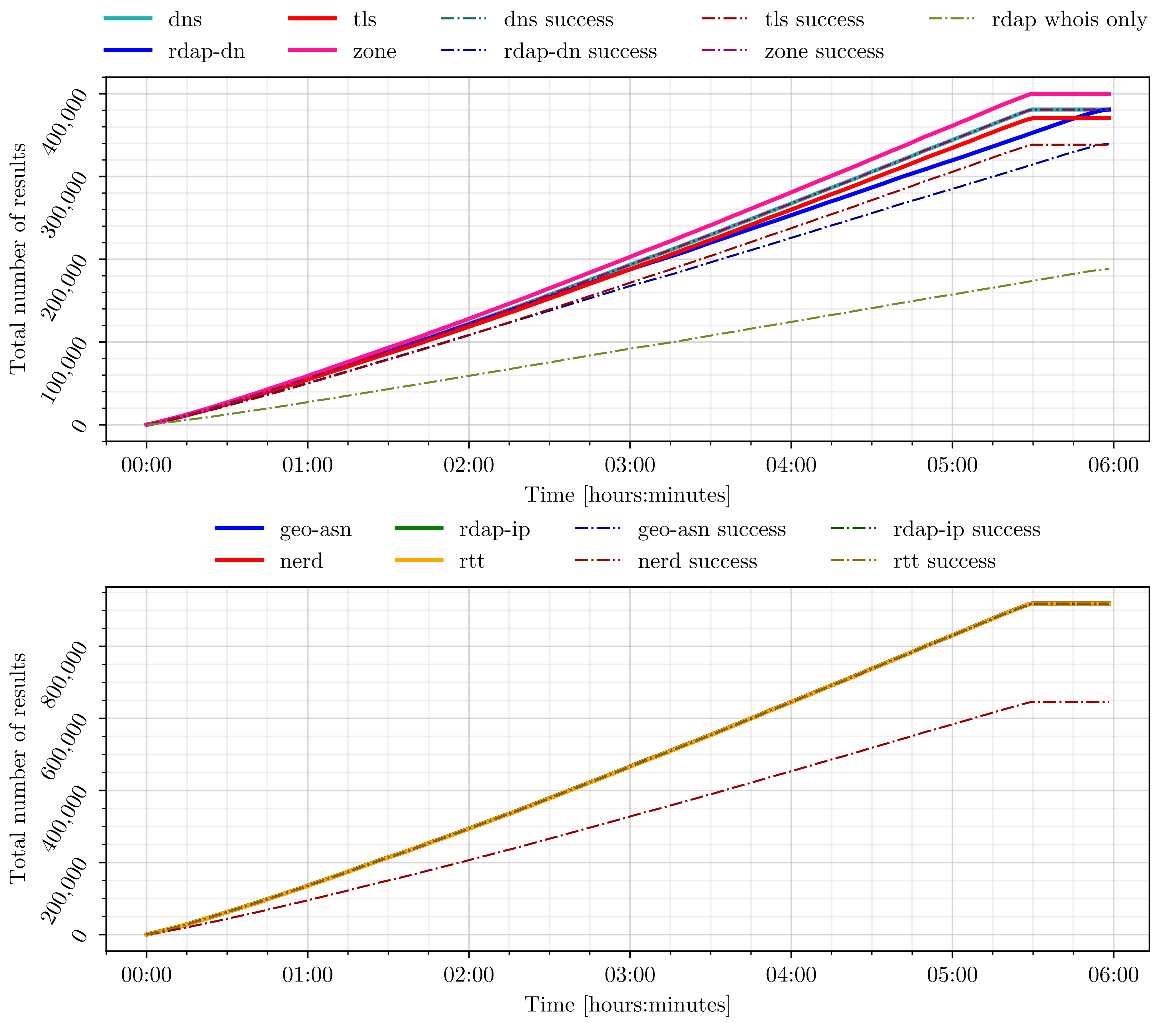


Figure 8.3: Number of responses from all collectors over time during experiment #3.

Experiment #4: 12 partitions

In the final experiment, the number of partitions was increased to $P_C = 12$, mainly to allow for additional scaling of the zone and DNS collectors. The Python-based collectors were set to $C_{Py} = 12$. The RTT collector was set to $C^c_{Py} = 128$, the zone and DNS collectors were set to $C^c_{Py} = 64$, and the RDAP collectors were set to $C^c_{Py} = 16$, as here, the rate limiter is likely slowing the collectors down anyway. The Java-based collectors were kept at two instances ($C_J = 2$) but the number of threads was increased to $C^t_J = 12$ to account for the additional input load. Table H.7 shows the per-collector metrics. Figure H.7 shows the response accumulation and the system resource usage measurements.

Findings:

- The average CPU usage was 42%; the peak was 69%. The average memory usage was 12.13 GiB; the peak was 13.88 GiB.
- The increase in the number of partitions positively affected the throughput of the zone collector, which managed to process all the entries in slightly over 3 hours.
- All the collectors except for RDAP-DN kept up with this throughput.
- The MnQdColT value decreased in the DNS collector, suggesting the increase in withstandable throughput is higher than the increase of the input load.
- Conversely, MnQdColT increased in the TLS, GEO-ASN, NERD, and RDAP-IP collectors, showing they are more utilised.

The same conclusions apply here as in the previous experiment. Here, the RDAP-DN collector is considerably more of a bottleneck, in comparison to the previous configuration. The memory usage trend is also similar: the 50% increase in collector instance count resulted in about a 53% increase in average memory usage.

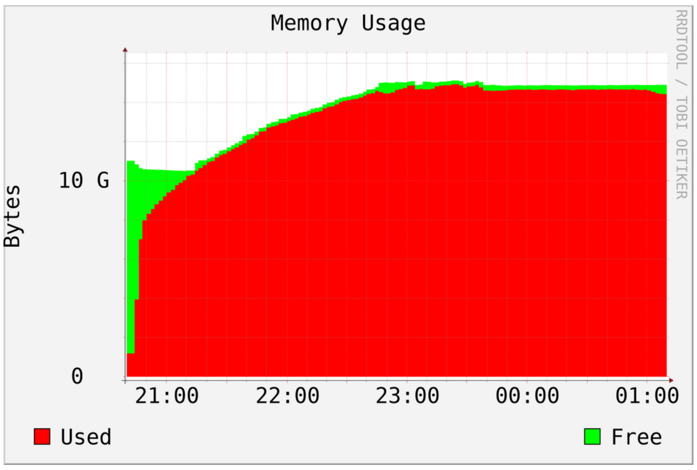


Figure 8.4: *Scanner* memory usage during experiment #4.

Collection errors

Tables H.2, H.4, H.6 and H.8 show the percentage of erroneous responses out of the total number of requests delivered to the collector (the "req" rows). For the DN-based collectors, they also list the percentages relative to 400,000 – the total number of input domain names in the system (the "all" rows). In the "WHOIS" column, the "req" values instead refer to the ratio of erroneous WHOIS responses to the total number of WHOIS requests only. The charts in Figures H.2, H.4, H.6 and H.8 show the breakdown of error codes reported in the collectors' responses in the respective experiments. The RTT collector never returned a non-zero status code, so it is not included in the charts or tables. The error rate of the GEO-ASN collector was always below 0.1% of the IP addresses, so it was also omitted from the charts for clarity.

Because the experiments used the same input data, the error rates were expected to be similar across them. The results confirm this: the success rates generally differ by less than 1%. Specifically, the Java-based TLS, NERD and GEO-ASN collectors were the most "stable", producing nearly the same result sets across all configurations. The charts suggest a slight rise in timeout errors in the TLS collector, but it is a statistically insignificant difference of units in a set of 400,000. The DNS collector was similarly consistent: here, the success rate was always over 99.94%, making the differences in the error composition also insignificant. The zone collector was consistent in the first three experiments but exhibited notably more timeout errors in the fourth. This is likely due to the increased number of requests targeted at only two remote DNS servers. The RDAP-IP collector was also surprisingly consistent, successfully processing over 99.7% of requests in all experiments, showing that the RIRs' RDAP servers do not enforce too strict rate limits.

RDAP-DN

The highest variability in the error composition was observed in the results of the RDAP-DN collector, which was expected, given that it contacts a high number of heterogeneous remote servers. Only 37% to 46% of the RDAP requests were successful, although when the WHOIS backup is considered, the overall success rate of the collector was 89% to 94%. With more parallelism, the overall error rate and the amount of timeout errors increased.

The error breakdown charts show RDAP and WHOIS errors separately. Note that the base count for the two bars is different: the RDAP error percentages are derived from the total number of responses with non-zero status codes, including the ones where WHOIS later succeeded (*i.e.* "Errors" + "WHOIS success"). The WHOIS error percentages are derived from the total number of failed WHOIS requests only (*i.e.* "Errors"). During evaluation, it was discovered that there had been an error in the WHOIS exception handling, which suppressed the actual result codes from both services when WHOIS was triggered. The charts represent these results as the "unknown" RDAP error and the `INTERNAL_ERROR` WHOIS error.

The results show that some RDAP servers do not use the HTTP 429 (Too Many Requests) status code when the rate limit is reached – for example, the server for the `.cz` TLD returns HTTP 503 (Service Unavailable). Such responses are represented by the `CANNOT_FETCH` result code instead of `RATE_LIMITED`. However, even when adding the amounts of the two errors, the total percentage of rate-limited queries (out of all requests) is about 11.4% in exp. #1, 13.8% in exp. #2, 12.9% in exp. #3, and 14.6% in exp. #4, showing that the amount of rate-limiting errors does not necessarily increase with scaling. In experiment #4, over 10% of the errors were timeouts, likely due to the increased load on the system.

Internal errors

The `INTERNAL_ERROR` result code was present in some results from the zone, DNS, RDAP-DN and RDAP-IP colllectors. However, further analysis shows that many of these are network-related and are only inappropriately reported. The following paragraphs describe the composition of internal errors in each collector based on the results from experiment #3.

Zone: All the internal errors were caused by the function that finds the IP addresses of the authoritative name servers. It did not handle the DNS SERVFAIL responses correctly, leading to an unhandled exception.

DNS: There were 36 internal errors: 33 were caused by the same problem as in the zone collector (when resolving the IP addresses for the CNAME, MX and NS records). The remaining three were caused by a UTF-8 decoding error when reading the TXT records. Additionally, there were 44 `OTHER_DNS_ERROR` errors, all caused by remote authoritative DNS servers not responding or returning invalid responses.

RDAP-DN: It is harder to determine the cause of the internal errors in the RDAP-DN collector as the errors were often caught at the top level of the collector, and some of the exceptions had no string representation. There were 47,645 responses marked as an internal error in total (incl. those represented in the charts as "unknown"), and 36,737 had a non-empty error string. Of these, 29,757 (81%) were caused by various network errors and 6,612 (18%) were caused by a TLS certificate verification failure. The other errors were related to the whodap library trying to process invalid RDAP responses.

RDAP-IP: Out of the total 567 internal errors here, 425 (75%) were caused by network errors (*e.g.* the server disconnecting without sending a response), and 142 (25%) were again caused by the whodap library processing.

Load on the broker

While the collectors were tested on a separate VM and optimised to process the data as fast as possible by maximising the use of the VM's resources, it is necessary to also determine how this affects the broker running on the other VM. A separate experiment was executed in a configuration similar to exp. #4, with 12 partitions and slightly tuned concurrency/threading settings. The experiment ran for about 3 hours and 20 minutes, but in the last 30 minutes, only the RDAP-DN collector was actively working. The graphs in Figure 8.5 show the CPU load on the two machines throughout the experiment. The average CPU usage on the *infra* VM was about 17%, and the average memory usage was about 3.83 GiB. The average CPU usage on the *scanner* VM was about 62%, and the memory usage was about 11.39 GiB. The load was nearly constant throughout the experiment, showing that the broker could handle the collectors' load without performance degradation. Figure H.9 shows the disk and network interface usage plots.

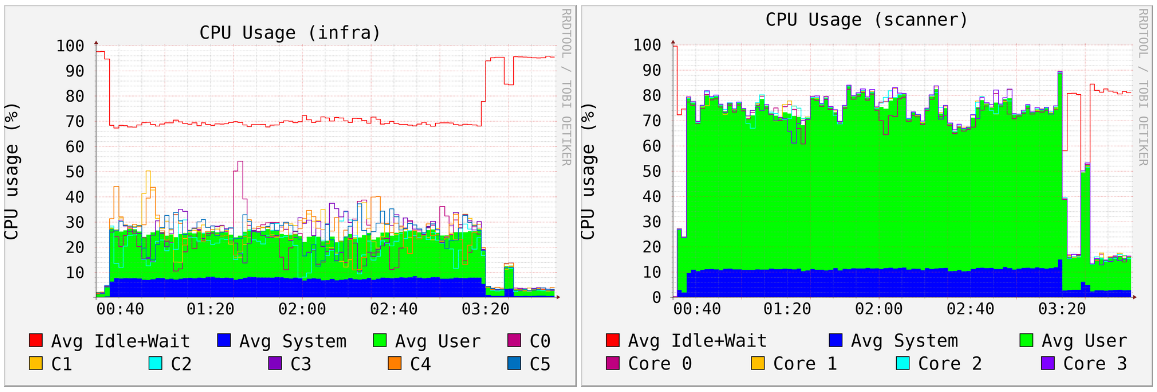


Figure 8.5: Comparison of the load on both VMs during collection.

Summary of the standalone collection experiments

The comprehensive experiments demonstrated that the implemented system can effectively manage horizontal scaling and adjust to high volumes of data with minimal errors. The experiments indicated that all collectors, except for RDAP-DN, maintained a consistent throughput. For instance, the collectors processed a list of 400,000 input entries in slightly over three hours when the number of partitions was increased to twelve. However, the RDAP-DN collector remains a bottleneck, reflecting its dependency on numerous and diverse remote servers. In production, the rate limiter settings and the throughput of the preceding stages must be adjusted to match the RDAP-DN collector's processing capabilities.

In the most demanding configuration, the system exhibited efficient resource utilisation with an average CPU usage of 42% and a peak of 69%. Memory usage was also within acceptable limits, with an average of 12.13 GiB and a peak of 13.88 GiB. However, a possible memory leak was detected as memory usage steadily increased over time. The load on the Kafka broker was also monitored, showing that the broker could handle the throughput of the collector pipeline.

The error rates remained low across most collectors. The RDAP-DN collector exhibited more variability in its success rates due to the heterogeneity of the remote servers it contacts. Nonetheless, the success rate ranged between 86% and 100%, even in the most resource-demanding configuration (not accounting for the NERD collector that does not support IPv6 addresses).

8.2 Data Merging Experiments

The data merger component was tested to evaluate the requirements on the system resources and to determine whether it makes sense to scale the process out when running on a single machine rather than to modify the number of processing threads. The experiment configurations differ in their settings of M and M^t.

Method

All the experiments were run on the same set of collected data. For each configuration, these steps were repeated:

1. Stop the running mergers.
2. Remove the intermediary topics made by KS, and recreate the *all_collected_data* topic.
3. Remove the mergers' state directories.
4. Adjust the merger scaling and threading options.
5. Change the mergers' application ID.
6. Start the merger services.

The merging results were not further analysed; only the total time and the system resource usage were monitored. The only services running on the system were the Kafka broker, the data merger, and Kafbat UI.

Observed metrics

- *Total processing time* (Tot, in minutes) is the time elapsed between the start of the merger services and the last produced message.
- *Mean processing time* (MnProcT, in milliseconds) is the average time of processing an input domain by the merger. It is the total processing time divided by 347,711 (the number of input domains with successful zone results).
- *Merger throughput* (MTput, in domain names per second) is the inverse of the mean processing time.
- *Average CPU usage* ($\overline{\text{CPU}}$, in %) is the across-cores average relative CPU time spent in userspace over the whole run.
- *Average memory usage* ($\overline{\text{Mem}}$, in GiB) is computed as the average memory usage (defined similarly to above) over the whole run.

Note that the MnProcT and MTput use the unique input domain names as the base, but the actual "event throughput" of the merger is higher, as it must process multiple events for each domain name. The number of events per domain name is highly variable, mainly because it depends on the number of IP addresses. The system resource metrics include CPU usage, memory usage, system load, and disk usage (which is important because the merger materialises its intermediary states both on the disk and in Kafka).

Input data

All the experiments below were executed on the same set of data collected in the last standalone collection experiment. The dataset consisted of:

- 401,207 zone results,
- 347,711 DNS and RDAP-DN results,
- 338,225 TLS results,
- 3,332,620 IP results (ca. 833,150 results from each IP-based collector or 9.58 IP addresses per DNS result).

In total, the merger processed 4,767,474 Kafka events in each run.

Configurations

The experiments were run with the following configurations:

1. $M = 4$, $M^t = 4$,
2. $M = 1$, $M^t = 16$,
3. $M = 1$, $M^t = 8$.

Exp.	Tot [min]	MnProcT [ms]	MTput [DN/s]	$\overline{\text{CPU}}$ [%]	$\overline{\text{Mem}}$ [GiB]
#1	33	5.69	175.61	**47**	36.63
#2	25	4.31	231.81	59	21.03
#3	**24**	**4.14**	**241.47**	57	**18.08**

Table 8.1: Metrics of the three merger runs. The best values are in bold.

Results

The metrics values for the three experiments are summarised in Table 8.1. The resource usage is plotted in Figure 8.6. The complete graphs plotting CPU usage, memory usage, system load, and disk usage during these tests are available in Appendix Section H.2. Surprisingly, the configuration #3 with the least "workers" had the highest throughput but not the lowest CPU usage. This suggests that the additional bookkeeping of multiple worker instances degrades the overall performance on these data. Also, when running on a single machine, it is way less memory-heavy to run the merger with multiple threads rather than in multiple instances.

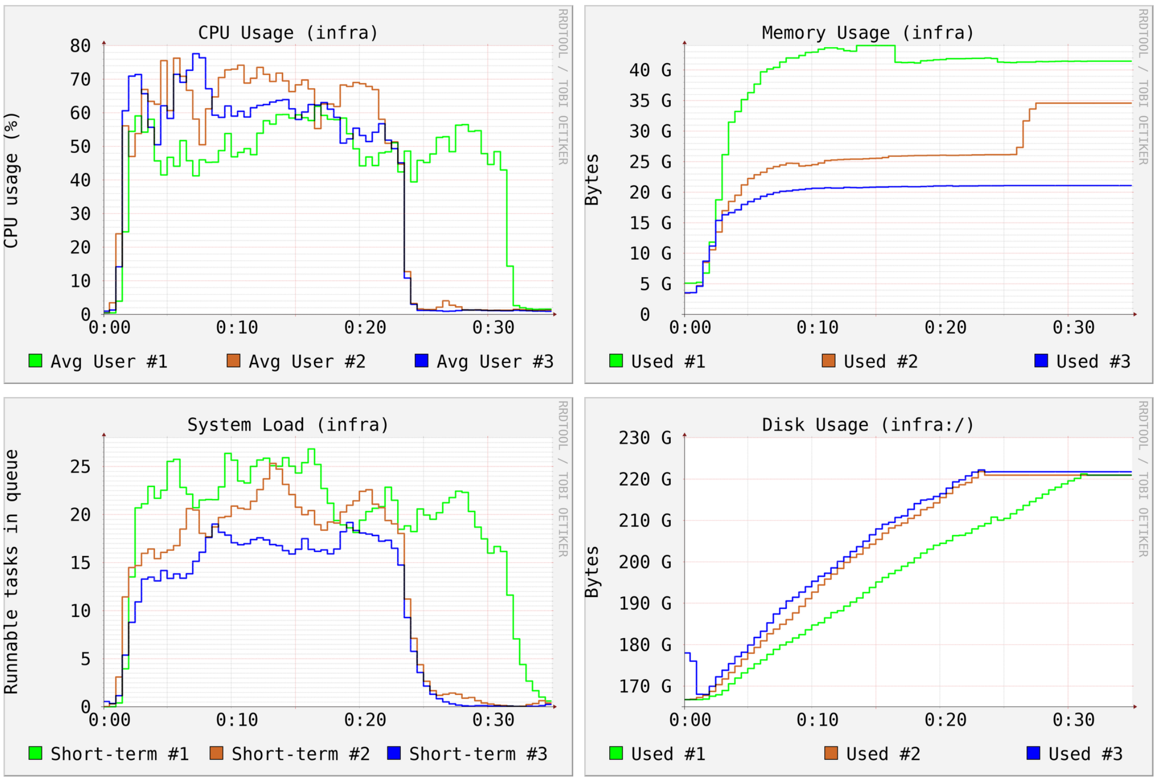


Figure 8.6: System metrics showing the differences between the three merger configurations.

Observe that the merging operation is quite resource-demanding in the default Kafka Streams configuration. The memory usage seems to be the limiting factor in the standalone collection use case. However, KS and RocksDB (used for the persistence of the state stores) both offer a variety of configuration options that can be used to tune the memory usage, as described in [11] and [21]. The need for low-level tuning is a drawback of the KS-based solution, as it requires a deep understanding of the underlying technologies. For example, both articles mention that it is advised to use an alternative memory allocator as the default one leads to increased memory consumption. This thesis does not focus on KS performance fine-tuning, but some attempts at memory usage optimisation were made for the real-time data processing experiments.

8.3 Standalone Feature Extraction Experiments

The feature extractor was also tested to evaluate the appropriate scaling in order to reach high throughputs. The effects of the instance count, batch size, and multiprocessing were evaluated.

Method

The steps repeated for each experiment were:

1. Stop the running extractor instances.
2. Adjust the extractor scaling and threading options.
3. Change the extractors' application ID.
4. Start the extractor services.

The feature extraction results were not further analysed. During the experiments, only the broker and Kafbat UI were running on the machine.

Input data

All the experiments were executed on the same set of data collected in the last standalone collection experiment. The input *all_collected_data* topic contained 419,204 events. In all experiments, the data were spread over 16 partitions.

Configurations

The experiments were run in a total of 11 configurations. For $E \in \{1, 4\}$, the experiments used a fixed batch size and varying amounts of the multiprocessing-based workers. For $E \in \{8, 16\}$, the experiments varied in batch sizes of 50, 100, and 200 but did not use multiprocessing. These configurations were selected with respect to the CPU and memory usage to keep the total process count around the number of partitions.

Observed metrics

In these experiments, only the average CPU and memory usage were metered. They are defined similarly to the previous experiments.

Results

The values of the metrics are summarised in Table 8.2. The complete graphs plotting the CPU and memory usage during the experiments are available in Appendix Section H.3. The results confirm that the extractor is CPU-intensive, with CPU usage consistently exceeding 60% in all configurations that employed scaling.

	Tot [min]	$\overline{\text{CPU}}$ [%]	$\overline{\text{Mem}}$ [GiB]	Tot [min]	$\overline{\text{CPU}}$ [%]	$\overline{\text{Mem}}$ [GiB]	Tot [min]	$\overline{\text{CPU}}$ [%]	$\overline{\text{Mem}}$ [GiB]
Inst. E	$E^b = 200,\ E^t = 4$			$E^b = 200,\ E^t = 8$			$E^b = 200,\ E^t = 16$		
1	22	46	**3.276**	22	**45**	3.954	19	46	5.162
4	10	78	6.520	11	78	8.671		———	
Inst. E	$E^b = 50$			$E^b = 100$			$E^b = 200$		
8	11	71	4.025	**9**	76	4.037	**9**	66	4.097
16	15	67	5.244	12	65	5.151	**9**	72	5.354

Table 8.2: System metrics during the feature extractor experiments.

The results also show that the multiprocessing-based scaling is not too effective in this configuration, as it did not lead to a decrease in the total processing time. This is quite surprising and should be further evaluated. The CPU usage does not increase with the number of workers, suggesting that the additional workers were actually not being utilised. It is possible that the de/serialisation process, performed in the main loop, takes more time than the actual feature extraction, suppressing the benefits of parallelism. This should be further investigated with the use of profiling.

The total performance did not increase considerably with scaling out; the total time was similar with 4 and 8 instances, and in some cases even higher with 16 instances. Apparently, the increased load on the system is preventing the workers from processing the data faster. Increasing the batch size seems to make processing slightly faster overall, although the difference is likely insignificant. The effects of batching could be increased by optimising the adopted transformations.

8.4 Real-time Data Processing Experiments

The next step was running all the system's components together in a real-world scenario to show how it copes with the required throughput.

Method

The original intention was to evaluate the system under full load from the CESNET academic network. However, due to technical issues on the CESNET side, the input (non-filtered) traffic rate was capped at 33 DN/s. As the final experiment showed, this yields a way-below-expected load of about 1.11 input domain names per second on the collector (after filtering). For this reason, synthetic data were added in some experiments to show the system's behaviour under higher loads.

The system configuration was tweaked based on the findings from the previous experiments. Five real-time experiments were executed. For the first two, the system was populated only with synthetic load – the domain names were loaded from a file at a given maximum rate. The third and fourth experiments combined synthetic load and traffic coming from the CESNET academic network in real time. In the final experiment, only the real network data were processed.

Configuration

The pipeline components were set up according to Table 8.3. All the experiments were executed with the same Kafka partitioning configuration: $P_C = P_M = P_E = 20$. The NERD batch size was 100, the extractor batch size was 100, and the batch timeout was 10 seconds. The merger configuration was slightly tweaked: it used the jemalloc memory allocator, the maximum cache size was set to 2 GiB total, and the commit interval was set to 2 seconds. Otherwise, the config matched the included configuration files. Note that the *infra* VM also ran the DomainRadar loader and classifier components and both database systems, which all contributed significantly to the overall system load.

Component	Instances	Concurrency	Threads	Timeout [s]
Zone	10	8	-	8
DNS	10	8	-	8
TLS	2	-	10	3
RDAP-DN	10	20	-	6
GEO-ASN	2	-	10	-
NERD	2	-	10	8
RDAP-IP	10	20	-	6
RTT	20	128	-	-
Extractor	5	-	1	-
Merger	1	-	20	-

Table 8.3: Configuration of the system for the real-time processing experiments.

Input data

Domain names from real traffic on the CESNET network were captured by the Suricata network analysis software and transferred to Elasticsearch. From there, they were loaded by the DomainRadar loader and pre-filter. The system published a batch of 20,000 records to Elasticsearch every 10 minutes, yielding an average rate of 33 DN/s. The loader was polling Elasticsearch for new records each second. It stored the domain names in the PostgreSQL database, from where they were polled by Kafka Connect. The source connector's poll rate was set to 1 second.

The input list of domain names for generating the synthetic load was created by merging the testing list from the standalone experiments with the Cloudflare top one million domain names list[5] from July 22, 2024 (see [80] for more information on the lists). The merged list, with 1,002,243 total entries, was then shuffled. The loader published $D \pm J_D$ domain names from the synthetic list each $R \pm J_R$ milliseconds – the time interval and DN count were randomised after each publish to better simulate real traffic. The randomisation was uniform, so the expected mean throughput was $E[T] = \frac{1000}{R} D$ domain names per second. Note that the actual throughput reported below was lower than the expectation, as the loader only provides an upper bound on the throughput.

The experiments were executed with the following configurations of synthetic load:

[5]Cloudflare domain rankings: `https://radar.cloudflare.com/domains`

- Exp. #1: $D = 4, J_D = 2, R = 600, J_R = 400 \quad \Rightarrow E[T] = 6.66$ DN/s,
- Exp. #2 and #3: $D = 5, J_D = 1, R = 460, J_R = 450 \quad \Rightarrow E[T] = 10.87$ DN/s,
- Exp. #4: $D = 5, J_D = 0, R = 250, J_R = 200 \quad \Rightarrow E[T] = 20.00$ DN/s.

Observed metrics

The complete results in Appendix Section H.4 contain the same tables and charts as in the standalone collector experiments, that is, the per-collector metrics of average throughput, average collection time and mean queued/collection time, the error rates breakdown table and charts, and the system resource usage charts. Some charts were intentionally omitted for experiments #1 and #2. The results table included in this section focuses only on the average and maximum CPU and memory usage.

Summary of the results

Run	Loaded	Time	$\overline{T}$	*infra*				*scanner*			
				CPU [%]		Mem [GiB]		CPU [%]		Mem [GiB]	
	[DN]	[h]	[DN/s]	avg	max	avg	max	avg	max	avg	max
Synthetic load only											
#1	291,896	18	4.50	27	92	21.22	23.98	12	20	9.33	10.43
#2	99,975	4.5	6.17	30	35	22.59	27.38	15	17	11.10	11.50
Synthetic load & real traffic											
#3	72,864	2.1	9.54	37	47	27.99	28.84	18	24	11.83	12.02
#4	340,070	12	7.89	34	52	32.33	35.50	16	24	9.58	10.71
Real traffic only											
#5	144,563	36	1.11	7	33	27.32	30.18	6	12	6.90	7.15

Table 8.4: System metrics during the real-time experiments.

The values of the resource usage metrics are summarised in Table 8.4. The results show that the system handled the throughput in all cases with a solid amount of headroom for increased load. The estimated common daily throughput was 300,000 domain names, corresponding to an average of 3.47 DN/s. Most experiments put more pressure on the collector (albeit in real traffic, the load will vary throughout the day), showing that it handles the requirement with ease.

In experiment #1, the data merger crashed for a short time after it tried to fetch a record that was larger than the default maximum size. This is apparent from Figure H.23 at around 18:30, where the CPU usage in the *infra* VM dropped. The merger was reconfigured to accept larger records and restarted.

The experiments confirm the increasing trend in memory usage both in the *infra* and *scanner* VMs. However, the measurements do not show a strong relation between the throughput or total number of processed items and memory usage, with the most memory-intensive experiment being the one with the lowest number of processed domain names. Even in the 36-hour experiment, the memory usage did not reach the limits of the VMs. Furthermore, manual observation showed that in the case of the *infra*, the increasing memory usage may have been caused by the MongoDB database system. At the end of experiment #4, it was using about 20 GiB of memory even when the system was idle. It is up to future tests to confirm or deny this hypothesis.

There were no notable problems in CPU usage. Here, the results show a stronger relation between throughput and CPU usage. A simple linear regression model over the data shows about a 3.48% increase in the average CPU usage on *infra* per 1 DN/s increase in the throughput. For *scanner*, the increase is about 1.42%. However, the amount of data here must be higher to draw firm conclusions.

The resource usage graphs in Figures H.27, H.30, and H.33 for experiments #3 to #5 show repetitive increases and decreases in both CPU and memory usage. These were caused by the CESNET infrastructure that inserted new domain names into the system in batches, causing periods of more intense work followed by periods where only the synthetic load was keeping the system busy.

Collection errors in the real-time operation

Tables H.10, H.12, and H.14 show the percentage of erroneous responses in experiments #3 to #5. They are accompanied by the charts in Figures H.26, H.29, and H.32, similar to the standalone collection tests. Note that for the two final experiments, error handling was adjusted according to the findings from the standalone collection experiments. Specifically, the WHOIS exception handling was fixed so that the RDAP-DN collector correctly reported the RDAP errors. The zone and DNS collectors were adjusted to handle the SERVFAIL responses correctly.

Overall, the structure of errors corresponds to findings from the standalone experiments. When comparing the real-time exp. #5 and the standalone collector exp. #3 (see Figure H.6), there is a notable increase in timeout errors across all the collectors, suggesting the need for a more relaxed timeout configuration. It was significant in the TLS collector: The total timeout error rate was 1.74% in the standalone experiment, while it reached 31.46% in the real-time operation.

It is interesting to compare the structure of errors between exp. #4, which included synthetic data from worldwide traffic, and exp. #5, which only processed the CESNET traffic from the Czech academic environment. For example, the total TLS timeout rate was significantly higher in #5 than in #4 (31.46% vs 9.04%), suggesting a trend in the configuration of local services. Conversely, the rate of the `NO_ENDPOINT` error from RDAP-DN, signalling that no RDAP service is provided for the TLD, was lower in #5 than in #4 (17.02% vs 23.69%).

In exp. #5, 9,920 zones (6.86%) were not found, which is not a large number overall but may seem surprising given that the names come from live traffic. Detailed analysis showed that out of these, 423 (4.3%) are "local" (*i.e.* without TLD or ending with `.local`), 149 (1.5%) contain characters that are not allowed in domain names, and 779 (7.9%) are likely DGA-generated (longer than 47 characters).

8.5 Discussion

The conducted experiments highlighted several key findings concerning the system's performance, reliability, and scalability. The system was tested in the two main use scenarios: standalone batch processing and real-time data collection for the DomainRadar system. The experiments utilised two virtual machines (VMs) within the CESNET network, each configured with specific hardware and software environments to manage different system components. The experiments demonstrated that the system could effectively handle high throughput with acceptable use of system resources. It managed the required throughput in the real-time experiments with sufficient headroom for increased load. The experiments also identified potential bottlenecks, particularly in the RDAP-DN collector and the data merger operation.

The experiments targeting the collectors evaluated the system's ability to scale horizontally by increasing Kafka partitions and the number of collector instances. They revealed that most collectors could handle the increased load effectively, with processing times decreasing as parallelism increased. However, the RDAP-DN collector consistently emerged as a bottleneck due to its reliance on numerous heterogeneous remote servers, leading to higher error rates. When processing a large batch of data, the collector was also notably slower than others, but its performance was sufficient in a real-time scenario. The experiments also showed that error rates remained low across the collectors and configurations, although they followed a slightly rising tendency when scaling. It is up to future experiments to observe how the system would behave if scaled out through multiple machines.

The measurements of the data merger component focused on determining whether instance scaling or increasing the number of processing threads would be more effective. The findings suggest that running the merger with multiple threads on a single machine is more efficient and less memory-intensive than running multiple instances. The merger processed almost 5 million records in under 30 minutes. However, the experiments highlighted the memory-intensive nature of the operation. While low-level tuning may optimise resource usage further, it is essential to implement a mechanism for removing old data from the state stores to prevent an indefinite growth of memory and disk usage. Nonetheless, the real-time experiments show that using the component in the current target production environment will be feasible until it is optimised further.

The feature extractor was tested to identify optimal scaling configurations. The results indicated that multiprocessing-based scaling was not as effective as expected, with additional workers not significantly reducing total processing time. This inefficiency was attributed to the de/serialisation process taking longer than the feature extraction itself, highlighting the need for further investigation and optimisation. Increasing the batch size slightly improved processing times, but the difference was minimal, suggesting that batching effects could be enhanced by optimising the transformations.

Overall, the evaluation demonstrated that the system could scale and adjust to high volumes of data with minimal errors. It was able to withstand the required throughput and handle the load effectively, with enough room for possible spikes in traffic. Some components, such as the RDAP-DN collector, the data merger, and possibly the feature extractor, offer room for further optimisation. The experiments highlighted the importance of tuning configurations to balance resource usage and performance, particularly in memory-intensive operations like data merging. Future work should address the identified bottlenecks and optimise the critical components' performance to ensure the system's reliability and efficiency in production environments.

Chapter 9

Conclusion

The research presented in this thesis addresses the issue of effective large-scale collection of information related to domain names based on sources such as the DNS, TLS handshakes, domain and IP address registration data, IP address reputation, and geolocation. The motivation for this work stemmed from my engagement in the FETA DomainRadar project. When designing the overall system, I brought forward the discussion on how to approach the problem of data processing. It became apparent quickly that the existing tooling would not be usable in production scenarios, necessitating evaluation of the expectations and introduction of a new data collection component.

We devised the high-level architecture of DomainRadar in the team. In this thesis, I evaluated the expected throughput and implemented the system's key component for data collection and storage. I showed that it handles the load while efficiently using available computing resources. I also integrated the research feature extraction process with the new system, ensuring the data are transformed into a form suitable for ML-based classification. As a part of this endeavour, I also significantly contributed to the overall design of the DomainRadar system. I devised how it would exchange data between its components and how both the operation data and evaluated data would be stored and processed.

I studied the common problems in distributed systems and data processing. I also researched the available technologies that could be used to implement the system. Based on my findings, I proposed a solution that utilises Apache Kafka, a distributed streaming platform, which provides a scalable and fault-tolerant solution for real-time data processing. Integrating PostgreSQL and MongoDB for data storage provided a reliable backend for managing the collected data. Kafka Connect facilitated smooth data transfer between the processing pipeline and the storage systems.

My experiments demonstrated the system's ability to scale out effectively, maximising resource usage to achieve higher throughput. In testing, the collectors maintained consistent performance across different configurations. When loaded with a batch of 400,000 domain names, the system collected all the data within 4 hours and 16 minutes, reaching the overall throughput of 22.65 domain names per second, while some of the components processed up to 75 records per second. In the real-time experiments, the system steadily processed data coming at a rate of up to 9.54 domain names per second with CPU usage well below 20% and an acceptable memory usage of below 12 GiB.

The system exhibited low error rates of between 0% and 20% across most components. Moreover, most errors were related to network limitations, which are solvable by vertical or horizontal scaling. The system's design ensured the errors were managed effectively and did not impact the overall function.

The thesis has demonstrated the feasibility of large-scale collection of domain-related data for use in malicious domain name classification. Its main contribution, the implementation of a large portion of the DomainRadar system, provides a solid foundation for future development and practical deployments, showing potential for enhancing the security of computer networks. Thinking beyond DomainRadar, the collection pipeline could find its uses in other projects in the network security field. The system's design and implementation can be adapted to other use cases that work with domain names or IP addresses, such as the enrichment of monitoring data or analysis of network logs.

Future work

The future development of the DomainRadar system can explore several promising directions to enhance performance and reliability. One area to consider is optimising the RDAP collectors by investigating the rate-limiting mechanisms of individual RDAP servers and fine-tuning local rate limiters to improve efficiency. Evaluating and optimising the reliability of individual components, particularly the Python-based collectors, is also crucial. Reimplementing all components in Java could be beneficial, given the superior performance of Java-based collectors observed in the experiments. Optimising the feature extraction process to reduce CPU usage and adopting a more efficient serialisation format than JSON could improve system performance and resource utilisation. The real-time data processing capabilities can be expanded by addressing the identified memory usage trends. These enhancements will contribute to the overall efficiency and reliability of the DomainRadar system, supporting its deployment in practical environments.

Bibliography

[1] Adams, T. *ScoutDNS Most Abused Top Level Domains List – October 2020*. 2020. Available at: https://www.scoutdns.com/most-abused-top-level-domains-list-october-scoutdns/.

[2] AFRINIC. *Registration Data Access Protocol (RDAP)*. Online. October 2023. Available at: https://afrinic.net/whois/rdap. [cit. 2024-01-10].

[3] Akidau, T.; Bradshaw, R.; Chambers, C.; Chernyak, S.; Fernández Moctezuma, R. J. et al. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proceedings of the VLDB Endowment*, 2015, vol. 8, p. 1792–1803. Available at: https://doi.org/10.14778/2824032.2824076.

[4] Anderson, B. and McGrew, D. Identifying Encrypted Malware Traffic with Contextual Flow Data. In: *Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security*. New York, NY, USA: Association for Computing Machinery, 2016, p. 35–46. AISec '16. ISBN 9781450345736. Available at: https://doi.org/10.1145/2996758.2996768.

[5] Antonakakis, M.; Perdisci, R.; Dagon, D.; Lee, W. and Feamster, N. Building a Dynamic Reputation System for DNS. In: *19th USENIX Security Symposium (USENIX Security 10)*. 2010, p. 273–290.

[6] Antonakakis, M.; Perdisci, R.; Lee, W.; II, N. V. and Dagon, D. Detecting Malware Domains at the Upper DNS Hierarchy. In: *20th USENIX Security Symposium (USENIX Security 11)*. San Francisco, CA: USENIX Association, August 2011. Available at: https://www.usenix.org/conference/usenix-security-11/detecting-malware-domains-upper-dns-hierarchy.

[7] Apache Software Foundation. *Apache Flink (v1.18.0). Stateful Stream Processing*. Online. 2023. Available at: https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/concepts/stateful-stream-processing/. [cit. 2024-01-07].

[8] Apache Software Foundation. *Apache Kafka. Documentation*. Online. 2024. Available at: https://kafka.apache.org/37/documentation.html. [cit. 2024-06-20].

[9] Apache Software Foundation. *Apache Kafka. Introduction*. Online. 2024. Available at: https://kafka.apache.org/intro. [cit. 2024-01-08].

[10] Apache Software Foundation. *Apache Kafka. Kafka Streams – Core Concepts*. Online. 2024. Available at:

https://kafka.apache.org/37/documentation/streams/core-concepts. [cit. 2024-06-20].

[11] Apache Software Foundation. *Apache Kafka. Kafka Streams – Memory Management.* Online. 2024. Available at: https://kafka.apache.org/37/documentation/streams/developer-guide/memory-mgmt. [cit. 2024-07-10].

[12] Apache Software Foundation. *Apache Kafka. Kafka Streams DSL.* Online. 2024. Available at: https://kafka.apache.org/37/documentation/streams/developer-guide/dsl-api. [cit. 2024-07-20].

[13] Apache Software Foundation. *Apache Kafka. Powered By.* Online. 2024. Available at: https://kafka.apache.org/powered-by. [cit. 2024-06-20].

[14] Arends, R.; Austein, R.; Larson, M.; Massey, D. and Rose, S. *DNS Security Introduction and Requirements* RFC 4033 (Proposed Standard). Request for Comments (RFC) 4033. Internet Engineering Task Force (IETF), march 2005. Available at: http://www.ietf.org/rfc/rfc4033.txt. Updated by RFCs 6014, 6840.

[15] Bartoš, V. NERD: Network Entity Reputation Database. In: *Proceedings of the 14th International Conference on Availability, Reliability and Security.* New York, NY, USA: Association for Computing Machinery, 2019. ARES '19. ISBN 9781450371643. Available at: https://doi.org/10.1145/3339252.3340512.

[16] Barut, O.; Luo, Y.; Zhang, T.; Li, W. and Li, P. *NetML: A Challenge for Network Traffic Analytics.* 2020.

[17] Bilge, L.; Kirda, E.; Kruegel, C. and Balduzzi, M. EXPOSURE: Finding malicious domains using passive DNS analysis. In: *NDSS.* 2011, p. 1–17.

[18] Blum, A.; Wardman, B.; Solorio, T. and Warner, G. Lexical Feature Based Phishing URL Detection Using Online Learning. In: *Proceedings of the 3rd ACM Workshop on Artificial Intelligence and Security.* New York, NY, USA: Association for Computing Machinery, 2010, p. 54–60. AISec '10. ISBN 9781450300889. Available at: https://doi.org/10.1145/1866423.1866434.

[19] Buber, E.; Diri, B. and Sahingoz, O. K. NLP Based Phishing Attack Detection from URLs. In: Abraham, A.; Muhuri, P. K.; Muda, A. K. and Gandhi, N., ed. *Intelligent Systems Design and Applications.* Cham: Springer International Publishing, 2018, p. 608–618. ISBN 978-3-319-76348-4.

[20] Bučko, F. *Klasifikácia doménových mien generovaných alogirtmami DGA.* Brno, CZ, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Hranický, R. Available at: https://www.vut.cz/studenti/zav-prace/detail/147226.

[21] Cadonna, B. *How to Tune RocksDB for Your Kafka Streams Application.* Online. Mar 2021. Available at: https://kafka.apache.org/37/documentation/streams/developer-guide/memory-mgmt. [cit. 2024-07-10].

[22] Canali, D.; Cova, M.; Vigna, G. and Kruegel, C. Prophiler: A Fast Filter for the Large-Scale Detection of Malicious Web Pages. In: *Proceedings of the 20th International Conference on World Wide Web*. New York, NY, USA: Association for Computing Machinery, 2011, p. 197–206. WWW '11. ISBN 9781450306324. Available at: https://doi.org/10.1145/1963405.1963436.

[23] Cersosimo, M. and Lara, A. Detecting Malicious Domains using the Splunk Machine Learning Toolkit. In: *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*. 2022, p. 1–6.

[24] CESNET. *CESNET*. Online. 2023. Available at: https://www.cesnet.cz/cesnet/?lang=en. [cit. 2023-12-28].

[25] CESNET. *Members*. Online. 2023. Available at: https://www.cesnet.cz/cesnet/members/?lang=en. [cit. 2023-12-28].

[26] Chatterjee, M. and Namin, A.-S. Detecting phishing websites through deep reinforcement learning. In: IEEE. *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*. 2019, vol. 2, p. 227–232.

[27] Chen, T. and Guestrin, C. XGBoost: A Scalable Tree Boosting System. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA: Association for Computing Machinery, 2016, p. 785–794. KDD '16. ISBN 978-1-4503-4232-2. Available at: https://doi.org/10.1145/2939672.2939785.

[28] Christou, O.; Pitropakis, N.; Papadopoulos, P.; McKeown, S. and Buchanan, W. Phishing URL Detection Through Top-level Domain Analysis: A Descriptive Approach. In: *Proceedings of the 6th International Conference on Information Systems Security and Privacy*. SCITEPRESS - Science and Technology Publications, 2020. Available at: http://dx.doi.org/10.5220/0008902202890298.

[29] Chronicle Cybersecurity. *Virustotal*. 2012. Available at: https://www.virustotal.com/.

[30] Cisco Systems. *PhishTank*. 2006. Available at: https://phishtank.org/.

[31] Cisco Systems. *Umbrella Popularity List*. Online. 2016. Available at: https://s3-us-west-1.amazonaws.com/umbrella-static/index.html. [cit. 2023-12-02].

[32] Colitti, L. *net: ipv6: Add IPv6 support to the ping socket*. May 2013. Available at: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git. Commit 6d0bfe22611602f36617bc7aa2ffa1bbb2f54c67.

[33] Confluent, Inc. *Confluent Cloud*. Online. 2024. Available at: https://www.confluent.io/confluent-cloud/. [cit. 2024-06-20].

[34] Confluent, Inc. *Confluent Platform*. Online. 2024. Available at: https://www.confluent.io/product/confluent-platform/. [cit. 2024-06-20].

[35] Crocker, D.; Hansen, T. and Kucherawy, M. *DomainKeys Identified Mail (DKIM) Signatures* RFC 6376 (Internet Standard). Request for Comments (RFC) 6376. Internet Engineering Task Force (IETF), september 2011. Available at: http://www.ietf.org/rfc/rfc6376.txt.

[36] Daigle, L. *WHOIS Protocol Specification* RFC 3912 (Draft Standard). Request for Comments (RFC) 3912. Internet Engineering Task Force (IETF), september 2004. Available at: http://www.ietf.org/rfc/rfc3912.txt.

[37] Darling, M.; Heileman, G.; Gressel, G.; Ashok, A. and Poornachandran, P. A lexical approach for classifying malicious URLs. In: *2015 International Conference on High Performance Computing & Simulation (HPCS)*. 2015, p. 195–202.

[38] Dean, J. and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA: USENIX Association, 2004, p. 137–150. Available at: https://doi.org/10.1145/1327452.1327492.

[39] Deng, L. and Yu, D. Deep Learning: Methods and Applications. *Foundations and Trends in Signal Processing*, 2014, vol. 7, 3–4, p. 197–387. ISSN 1932-8346. Available at: http://dx.doi.org/10.1561/2000000039.

[40] Dierks, T. and Rescorla, E. *The Transport Layer Security (TLS) Protocol Version 1.2* RFC 5246 (Proposed Standard). Request for Comments (RFC) 5246. Internet Engineering Task Force (IETF), august 2008. Available at: http://www.ietf.org/rfc/rfc5246.txt. Updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685.

[41] Drichel, A.; Drury, V.; Brandt, J. von and Meyer, U. Finding phish in a haystack: A pipeline for phishing classification on certificate transparency logs. In: *Proceedings of the 16th International Conference on Availability, Reliability and Security*. 2021, p. 1–12.

[42] Drichel, A.; Faerber, N. and Meyer, U. First step towards explainable DGA multiclass classification. In: *Proceedings of the 16th International Conference on Availability, Reliability and Security*. Association for Computing Machinery, 2021, p. 1–13. ISBN 978-1-4503-9051-4. Available at: https://doi.org/10.48550/arXiv.2106.12336.

[43] Eastlake, D. *Transport Layer Security (TLS) Extensions: Extension Definitions* RFC 6066 (Proposed Standard). Request for Comments (RFC) 6066. Internet Engineering Task Force (IETF), january 2011. Available at: http://www.ietf.org/rfc/rfc6066.txt.

[44] Ecma International. *ECMA-404: The JSON Data Interchange Syntax*. Ecma Standard 404, 2nd ed. December 2017. Available at: https://www.ecma-international.org/publications-and-standards/standards/ecma-404/.

[45] Elastic. *Elasticsearch: The Official Distributed Search & Analytics Engine*. Online. 2023. Available at: https://www.elastic.co/elasticsearch. [cit. 2023-12-01].

[46] Elz, R. and Bush, R. *Clarifications to the DNS Specification* RFC 2181 (Proposed Standard). Request for Comments (RFC) 2181. Internet Engineering Task Force (IETF), july 1997. Available at: http://www.ietf.org/rfc/rfc2181.txt. Updated by RFCs 4035, 2535, 4343, 4033, 4034, 5452.

[47] Feng, J.; Zhang, Y. and Qiao, Y. A Detection Method for Phishing Web Page Using DOM-Based Doc2Vec Model. *Journal of Computing and Information Technology*, july 2020, vol. 28, p. 19–31. Available at: http://cit.fer.hr/index.php/CIT/article/view/4899.

[48] García, S.; Hynek, K.; Vekshin, D.; Čejka, T. and Wasicek, A. Large Scale Measurement on the Adoption of Encrypted DNS. *CoRR*, 2021, abs/2107.04436. Available at: https://arxiv.org/abs/2107.04436.

[49] Hajaj, C.; Hason, N. and Dvir, A. Less is more: Robust and novel features for malicious domain detection. *Electronics*. MDPI, 2022, vol. 11, no. 6, p. 969.

[50] Hamroun, C.; Amamou, A.; Haddadou, K.; Haroun, H. and Pujolle, G. A Review On Lexical Based Malicious Domain Name Detection Methods. In: *2022 6th Cyber Security in Networking Conference (CSNet)*. 2022, p. 1–7. Available at: https://ieeexplore.ieee.org/abstract/document/9955618.

[51] Hao, S.; Feamster, N. and Pandrangi, R. Monitoring the initial DNS behavior of malicious domains. In: *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*. New York, NY, USA: Association for Computing Machinery, 2011, p. 269–278. IMC '11. ISBN 9781450310130. Available at: https://doi.org/10.1145/2068816.2068842.

[52] Harel, B. *AsyncIO Rate Limiter for Python*. Online. 2023. Available at: https://asynciolimiter.readthedocs.io/en/latest/. [cit. 2024-06-29].

[53] Hason, N.; Dvir, A. and Hajaj, C. Robust malicious domain detection. In: Springer. *Cyber Security Cryptography and Machine Learning: Fourth International Symposium, CSCML 2020, July 2–3, 2020, Proceedings 4*. 2020, p. 45–61.

[54] Hoffman, P.; Sullivan, A. and Fujiwara, K. *DNS Terminology* RFC 8499 (Best Current Practice). Request for Comments (RFC) 8499. Internet Engineering Task Force (IETF), march 2019. Available at: http://www.ietf.org/rfc/rfc8499.txt.

[55] Hoffman, P. E. *DNS Security Extensions (DNSSEC)* RFC 9364 (Best Current Practice). Request for Comments (RFC) 9364. Internet Engineering Task Force (IETF), february 2023. Available at: http://www.ietf.org/rfc/rfc9364.txt.

[56] Hollenbeck, S. and Newton, A. *JSON Responses for the Registration Data Access Protocol (RDAP)* RFC 9083 (Internet Standard). Request for Comments (RFC) 9083. Internet Engineering Task Force (IETF), june 2021. Available at: http://www.ietf.org/rfc/rfc9083.txt.

[57] Horák, A. *Detekce škodlivých domén na základě externích zdrojů dat*. Brno, CZ, 2023. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Hranický, R. Available at: https://www.vut.cz/studenti/zav-prace/detail/146391.

[58] Hranický, R.; Horák, A.; Polišenský, J.; Pouč, P. and Ondryáš, O. *Phishing and Benign Domain Dataset (DNS, IP, WHOIS/RDAP, TLS, GeoIP)*. Zenodo, september 2023. Available at: https://doi.org/10.5281/zenodo.8364668.

[59] Hranický, R.; Horák, A.; Polišenský, J.; Jeřábek, K. and Ryšavý, O. Unmasking the Phishermen: Phishing Domain Detection with Machine Learning and Multi-Source Intelligence. In: *NOMS 2024-2024 IEEE Network Operations and Management Symposium*. 2024, p. 1–5.

[60] Hueske, F. and Kalavri, V. *Stream Processing with Apache Flink*. 1st ed. Sebastopol, CA, USA: O'Reilly Media, 2019. ISBN 978-1-491-97429-2.

[61] ICANN. *GTLD RDAP Profile*. Online. February 2019. Available at: https://www.icann.org/gtld-rdap-profile. [cit. 2023-01-07].

[62] ICANN. *2013 Registrar Accreditations Agreement*. Online. June 2023. Available at: https://www.icann.org/resources/pages/registrar-accreditation-agreement-2023-06-08-en. [cit. 2024-01-07].

[63] ICANN. *2023 Global Amendments to the Base gTLD Registry Agreement (RA), Specification 13, and 2013 Registrar Accreditation Agreement (RAA)*. Online. 2023. Available at: https://www.icann.org/resources/pages/global-amendment-2023-en. [cit. 2023-12-20].

[64] ICANN GAC. *WHOIS and Data Protection*. Online. October 2021. Available at: https://gac.icann.org/activity/whois-and-data-protection. [cit. 2023-01-07].

[65] Jeyaraj, R.; Pugalendhi, G. and Paul, A. *Big Data with Hadoop MapReduce: A Classroom Approach*. 1st ed. Apple Academic Press, 2021. 426 p. ISBN 9781774634844.

[66] Kitterman, S. *Sender Policy Framework (SPF) for Authorizing Use of Domains in Email, Version 1* RFC 7208 (Proposed Standard). Request for Comments (RFC) 7208. Internet Engineering Task Force (IETF), april 2014. Available at: http://www.ietf.org/rfc/rfc7208.txt. Updated by RFC 7372.

[67] Kshemkalyani, A. D. and Singhal, M. *Distributed Computing: Principles, Algorithms, and Systems*. 1st ed. Cambridge University Press, 2008. 756 p. ISBN 9780521876346.

[68] Kucherawy, M. and Zwicky, E. *Domain-based Message Authentication, Reporting, and Conformance (DMARC)* RFC 7489 (Informational). Request for Comments (RFC) 7489. Internet Engineering Task Force (IETF), march 2015. Available at: http://www.ietf.org/rfc/rfc7489.txt.

[69] Kulikov, V. *net: ipv4: add IPPROTO_ICMP socket kind*. May 2011. Available at: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git. Commit c319b4d76b9e583a5d88d6bf190e079c4e43213d.

[70] Kuyama, M.; Kakizaki, Y. and Sasaki, R. Method for detecting a malicious domain by using WHOIS and DNS features. In: *3rd International Conference on Digital Security and Forensics*. 2016, vol. 74.

[71] LACNIC. *Accessing RDAP*. Online. 2024. Available at: https://www.lacnic.net/676/2/lacnic/request-rdap-access. [cit. 2024-01-10].

[72] Let's Encrypt. *Let's Encrypt Stats. Percentage of Web Pages Loaded by Firefox Using HTTPS*. Online. 2023. Available at: https://letsencrypt.org/stats/#percent-pageloads. [cit. 2023-12-21].

[73] Ligthbend, Inc.. *Akka Documentation. Basics and working with Flows*. Online. 2024. Available at: https://doc.akka.io/docs/akka/current/stream/stream-flows-and-basics.html. [cit. 2024-05-20].

[74] Ligthbend, Inc.. *Akka Documentation. Getting Started Guide*. Online. 2024. Available at: https://doc.akka.io/docs/akka/current/typed/guide/index.html. [cit. 2024-05-20].

[75] Lin, M.-S.; Chiu, C.-Y.; Lee, Y.-J. and Pao, H.-K. Malicious URL filtering – A big data application. In: *2013 IEEE International Conference on Big Data*. 2013, p. 589–596.

[76] Lin, T.; Capecci, D. E.; Ellis, D. M.; Rocha, H. A.; Dommaraju, S. et al. Susceptibility to Spear-Phishing Emails: Effects of Internet User Demographics and Email Content. *ACM Trans. Comput.-Hum. Interact.* New York, NY, USA: Association for Computing Machinery, jul 2019, vol. 26, no. 5. ISSN 1073-0516. Available at: https://doi.org/10.1145/3336141.

[77] Liu, Z.; Zeng, Y.; Zhang, P.; Xue, J.; Zhang, J. et al. An Imbalanced Malicious Domains Detection Method Based on Passive DNS Traffic Analysis. *Security and Communication Networks*. Hindawi, 2018, vol. 2018, p. 7. Available at: https://www.hindawi.com/journals/scn/2018/6510381/.

[78] LOVOO and Goka community. *Goka*. Online. 2024. Available at: https://github.com/lovoo/goka. [cit. 2024-06-20].

[79] Maas, G. and Garillot, F. *Stream Processing with Apache Spark*. 1st ed. Sebastopol, CA, USA: O'Reilly Media, 2019. ISBN 978-1-491-94424-0.

[80] Martinho, C. and Zejnilovic, S. *Goodbye, Alexa. Hello, Clouflare Radar Domain Rankings*. *The Cloudflare Blog*. Online. Sep 2022. Available at: https://blog.cloudflare.com/radar-domain-rankings. [cit. 2024-07-22].

[81] Michaelson, G. New RDAP profile to improve cross-RIR consistency. *APNIC Blog* online, february 2021. Available at: https://blog.apnic.net/2021/02/16/new-rdap-profile-to-improve-cross-rir-consistency/. [cit. 2023-01-10].

[82] Mishra, S. and Soni, D. Smishing Detector: A security model to detect smishing through SMS content analysis and URL behavior analysis. *Future Generation Computer Systems*, 2020, vol. 108, p. 803–815. ISSN 0167-739X. Available at: https://www.sciencedirect.com/science/article/pii/S0167739X19318758.

[83] MISP Contributors. *Quick Start – User guide of MISP intelligence sharing platform*. Online. January 2023. Available at: https://www.circl.lu/doc/misp/quick-start/. [cit. 2023-12-10].

[84] Mockapetris, P. *Domain names - concepts and facilities* RFC 1034 (Internet Standard). Request for Comments (RFC) 1034. Internet Engineering Task Force (IETF), november 1987. Available at: http://www.ietf.org/rfc/rfc1034.txt. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936.

[85] Mockapetris, P. *Domain names - implementation and specification* RFC 1035 (Internet Standard). Request for Comments (RFC) 1035. Internet Engineering Task Force (IETF), november 1987. Available at: http://www.ietf.org/rfc/rfc1035.txt. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2673, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966, 6604.

[86] MongoDB, Inc. *MongoDB Community Edition*. Online. 2024. Available at: https://www.mongodb.com/products/self-managed/community-edition. [cit. 2024-06-25].

[87] Mozilla. *Public Suffix List*. Online. 2022. Available at: https://publicsuffix.org/. [cit. 2023-12-21].

[88] Newton, A.; Ellacott, B. and Kong, N. *HTTP Usage in the Registration Data Access Protocol (RDAP)* RFC 7480 (Proposed Standard). Request for Comments (RFC) 7480. Internet Engineering Task Force (IETF), march 2015. Available at: http://www.ietf.org/rfc/rfc7480.txt.

[89] Newton, A. and Hollenbeck, S. *Registration Data Access Protocol (RDAP) Query Format* RFC 7482 (Proposed Standard). Request for Comments (RFC) 7482. Internet Engineering Task Force (IETF), march 2015. Available at: http://www.ietf.org/rfc/rfc7482.txt.

[90] Niakanlahiji, A.; Chu, B.-T. and Al Shaer, E. PhishMon: A Machine Learning Framework for Detecting Phishing Webpages. In: *2018 IEEE International Conference on Intelligence and Security Informatics (ISI)*. 2018, p. 220–225. Available at: https://ieeexplore.ieee.org/document/8587410.

[91] Nominet. *Our domain names: Which .uk is right for you?* Online. 2023. Available at: https://theukdomain.uk/uk-domain-family/. [cit. 2023-12-28].

[92] NRO. *NRO RDAP Profile* Online. RDAP. The Number Resource Organization (NRO), january 2021. Available at: https://bitbucket.org/nroecg/nro-rdap-profile/src/master/nro-rdap-profile.txt.

[93] NumFOCUS, Inc. *About pandas*. Online. 2024. Available at: https://pandas.pydata.org/about/index.html. [cit. 2024-07-01].

[94] NumFOCUS, Inc. *IO tools. Pandas 2.2.2 documentation*. Online. 2024. Available at: https://pandas.pydata.org/docs/user_guide/io.html. [cit. 2024-07-01].

[95] Ongaro, D. and Ousterhout, J. In search of an understandable consensus algorithm. In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USA: USENIX Association, 2014, p. 305–320. USENIX ATC'14. ISBN 9781931971102.

[96] OpenPhish. *OpenPhish*. 2014. Available at: https://openphish.com/.

[97] OpenSSL Project Authors. *SSL_CTX_set_security_level. OpenSSL*. Online. 2024. Available at: https://www.openssl.org/docs/man1.1.1/man3/SSL_CTX_set_security_level.html. [cit. 2024-06-30].

[98] Oracle. *Virtual Threads. Java SE 21 Core Libraries*. Online. 2024. Available at: https://docs.oracle.com/en/java/javase/21/core/virtual-threads.html. [cit. 2024-07-02].

[99] Palaniappan, G.; Sangeetha, S.; Rajendran, B.; Goyal, S.; Bindhumadhava, B. et al. Malicious domain detection using machine learning on domain name features, host-based features and web-based features. *Procedia Computer Science*. Elsevier, 2020, vol. 171, p. 654–661.

[100] Passerini, E.; Paleari, R.; Martignoni, L. and Bruschi, D. FluXOR: Detecting and Monitoring Fast-Flux Service Networks. In: Zamboni, D., ed. *Detection of Intrusions and Malware, and Vulnerability Assessment*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, p. 186–206. ISBN 978-3-540-70542-0.

[101] Perdisci, R.; Corona, I. and Giacinto, G. Early detection of malicious flux networks via large-scale passive DNS traffic analysis. *IEEE Transactions on Dependable and Secure Computing*. IEEE, 2012, vol. 9, no. 5, p. 714–726.

[102] Pienta, D.; Jason, B. T. and Johnston, A. Protecting a whale in a sea of phish. *Journal of Information Technology*, september 2020, vol. 35, no. 3, p. 214–231. Available at: https://www.proquest.com/scholarly-journals/protecting-whale-sea-phish/docview/2439229129/se-2.

[103] Postel, J. *Domain Name System Structure and Delegation* RFC 1591 (Informational). Request for Comments (RFC) 1591. Internet Engineering Task Force (IETF), march 1994. Available at: http://www.ietf.org/rfc/rfc1591.txt.

[104] Postel, J. and Reynolds, J. *Domain requirements* RFC 920. Request for Comments (RFC) 920. Internet Engineering Task Force (IETF), october 1984. Available at: http://www.ietf.org/rfc/rfc920.txt.

[105] Prieto, I.; Magaña, E.; Morato, D. and Izal, M. Botnet Detection based on DNS Records and Active Probing. In: *Proceedings of the International Conference on Security and Cryptography*. IEEE, January 2011, p. 307–316. ISBN 978-989-8425-71-3.

[106] Python Software Foundation. *multiprocessing – Process-based parallelism. Python 3.12.4 documentation: Contexts and start methods*. Online. Jul 2024. Available at: https://docs.python.org/3/library/multiprocessing.html#contexts-and-start-methods. [cit. 2024-07-20].

[107] Python Wiki Contributors. *GlobalInterpreterLock*. Online. 2020. Available at: https://wiki.python.org/moin/GlobalInterpreterLock. [cit. 2023-12-20].

[108] Quix Analytics, Ltd. *Processing & Transforming Data. Quix documentation.* Online. 2024. Available at: https://quix.io/docs/quix-streams/processing.html. [cit. 2024-06-20].

[109] Rahbarinia, B.; Perdisci, R. and Antonakakis, M. Efficient and Accurate Behavior-Based Tracking of Malware-Control Domains in Large ISP Networks. *ACM Trans. Priv. Secur.* New York, NY, USA: Association for Computing Machinery, 2016, vol. 19, no. 2. ISSN 2471-2566. Available at: https://doi.org/10.1145/2960409.

[110] Redpanda Data Inc. *Features and capabilities for getting your real-time game on. Redpanda.* Online. 2024. Available at: https://redpanda.com/platform-capabilities. [cit. 2024-06-20].

[111] RIPE NCC. *RIPE Database Docs. Registration Data Access Protocol (RDAP).* Online. 2023. Available at: https://apps.db.ripe.net/docs/How-to-Query-the-RIPE-Database/Registration-Data-Access-Protocol/. [cit. 2024-01-10].

[112] Robinhood Markets, Inc. and faust-streaming community. *Faust 0.11.0 documentation. Introducing Faust.* Online. 2024. Available at: https://faust-streaming.github.io/faust/introduction.html. [cit. 2024-06-20].

[113] Sadique, F.; Kaul, R.; Badsha, S. and Sengupta, S. An automated framework for real-time phishing URL detection. In: IEEE. *2020 10th Annual Computing and Communication Workshop and Conference (CCWC).* 2020, p. 0335–0341. Available at: https://ieeexplore.ieee.org/document/9031269.

[114] Shi, Y.; Chen, G. and Li, J. Malicious domain name detection based on extreme machine learning. *Neural Processing Letters.* Springer, 2018, vol. 48, p. 1347–1357.

[115] Singh, A. and Goyal, N. A comparison of machine learning attributes for detecting malicious websites. In: IEEE. *2019 11th International Conference on Communication Systems & Networks (COMSNETS).* 2019, p. 352–358.

[116] Singh, J. ARIN Achieves NRO RDAP Profile Conformance. *ARIN Blog* online, march 2022. Available at: https://www.arin.net/blog/2022/03/09/nro-rdap-profile-conformance/. [cit. 2023-01-10].

[117] Stubbs, A. *Introducing the Confluent Parallel Consumer.* Online. 2020. Available at: https://www.confluent.io/blog/introducing-confluent-parallel-message-processing-client/. [cit. 2024-06-20].

[118] Technologická agentura ČR. *Flow-based Encrypted Traffic Analysis.* Online. 2022. Available at: https://starfos.tacr.cz/en/projekty/VJ02010024. [cit. 2023-12-01].

[119] The PostgreSQL Global Development Group. *About. PostgreSQL.* Online. 2024. Available at: https://www.postgresql.org/about/contact/. [cit. 2024-06-25].

[120] Thomson, S.; Huitema, C.; Ksinant, V. and Souissi, M. *DNS Extensions to Support IP Version 6* RFC 3596 (Draft Standard). Request for Comments (RFC)

3596. Internet Engineering Task Force (IETF), october 2003. Available at: http://www.ietf.org/rfc/rfc3596.txt.

[121] Torroledo, I.; Camacho, L. D. and Bahnsen, A. C. Hunting malicious TLS certificates with deep neural networks. In: *Proceedings of the 11th ACM workshop on Artificial Intelligence and Security*. 2018, p. 64–73.

[122] Tuan, T. A.; Long, H. V. and Taniar, D. On Detecting and Classifying DGA Botnets and their Families. *Computers & Security*, 2022, vol. 113, p. 102549. ISSN 0167-4048. Available at: https://www.sciencedirect.com/science/article/pii/S0167404821003734.

[123] Visser, M. *FLIP-265 Deprecate and remove Scala API support* FLIP-265. Flink Improvement Proposal (FLIP) 265. Apache Software Foundation, october 2022. Available at: https://cwiki.apache.org/confluence/display/FLINK/FLIP-265+Deprecate+and+remove+Scala+API+support. [cit. 2024-01-07].

[124] Yadav, S.; Reddy, A. K. K.; Reddy, A. L. N. and Ranjan, S. Detecting Algorithmically Generated Domain-Flux Attacks With DNS Traffic Analysis. *IEEE/ACM Transactions on Networking*, 2012, vol. 20, no. 5, p. 1663–1677.

[125] Zhao, H.; Chang, Z.; Bao, G. and Zeng, X. Malicious Domain Names Detection Algorithm Based on N-Gram. *Journal of Computer Networks and Communications*, 2019, vol. 2019.

[126] Zhauniarovich, Y.; Khalil, I.; Yu, T. and Dacier, M. A Survey on Malicious Domains Detection through DNS Data Analysis. *ACM Comput. Surv.* New York, NY, USA: Association for Computing Machinery, jul 2018, vol. 51, no. 4. ISSN 0360-0300. Available at: https://doi.org/10.1145/3191329.

[127] Zhou, L.; Kong, N.; Shen, S.; Sheng, S. and Servin, A. *Inventory and Analysis of WHOIS Registration Objects* RFC 7485 (Informational). Request for Comments (RFC) 7485. Internet Engineering Task Force (IETF), march 2015. Available at: http://www.ietf.org/rfc/rfc7485.txt.

[128] Zieni, R.; Massari, L. and Calzarossa, M. C. Phishing or Not Phishing? A Survey on the Detection of Phishing Websites. *IEEE Access*, 2023, vol. 11, p. 18499–18519.

[129] Zomaya, A. Y. and Sakr, S., ed. *Handbook of Big Data Technologies*. 1st ed. Cham: Springer, 2017. ISBN 978-3-319-49339-8.

Appendix A

CESNET Domains Analysis

Table A.1 shows the top 15 individual domain names, effective third-level domains and effective second-level domains in the CESNET sample (discussed in Section 5.2) according to the average number of resolutions per day. In the aggregates, the number of resolutions for all domains in a given e3LD/eSLD was added and divided by the number of days in which the e3LD/eSLD was present.

Table A.2 then shows the top 15 e3LDs and eSLDs according to the average number of subdomains observed for each. In the aggregates, the number of subdomains in a given e3LD/eSLD was added and divided by the number of days in which the e3LD/eSLD was present.

Table A.1: Top domain names/e3LDs/eSLDs by request count.

#	**Domain name**	**Avg. req. per day**
	Top domain names (whole)	
1	gateway.icloud.com	9,607,822
2	v10.events.data.microsoft.com	8,596,945
3	graph.facebook.com	7,103,137
4	outlook.office365.com	6,986,901
5	login.microsoftonline.com	6,337,459
6	settings-win.data.microsoft.com	5,494,072
7	web.facebook.com	4,480,572
8	graph.microsoft.com	4,174,902
9	eu-v20.events.data.microsoft.com	3,986,292
10	is.cuni.cz	3,767,726
11	self.events.data.microsoft.com	3,701,912
12	www.google.com	3,545,446
13	assets.msn.com	3,501,037
14	client.wns.windows.com	3,474,030
15	edge-mqtt.facebook.com	3,396,467

#	Domain name	Avg. req. per day
Top effective third-level domains		
1	data.microsoft.com	32,615,060
2	mp.microsoft.com	15,055,218
3	gateway.icloud.com	9,613,051
4	officeapps.live.com	8,291,242
5	graph.facebook.com	7,103,137
6	outlook.office365.com	6,986,902
7	teams.microsoft.com	6,484,081
8	login.microsoftonline.com	6,338,988
9	itunes.apple.com	5,529,952
10	web.facebook.com	4,480,572
11	ls.apple.com	4,331,743
12	smartscreen.microsoft.com	4,286,834
13	is.cuni.cz	4,178,080
14	graph.microsoft.com	4,174,902
15	wns.windows.com	3,654,477
Top effective second-level domains		
1	microsoft.com	77,163,962
2	apple.com	24,529,528
3	facebook.com	22,744,655
4	googleapis.com	16,093,404
5	seznam.cz	16,013,777
6	icloud.com	15,150,699
7	live.com	14,968,418
8	google.com	13,842,479
9	msn.com	9,382,622
10	cuni.cz	7,981,147
11	office365.com	7,336,673
12	tiktokv.com	6,792,486
13	tiktokcdn.com	6,435,627
14	microsoftonline.com	6,378,520
15	bing.com	6,278,633

Table A.2: Top e3LDs/eSLDs by subdomain count.

#	**Domain name**	**Avg. subdomains per day**
Top effective third-level domains		
1	safeframe.googlesyndication.com	452,592
2	measure.office.com	185,939
3	init.cedexis-radar.net	49,663
4	metric.gstatic.com	31,924
5	clo.footprintdns.com	17,280
6	nuid.imrworldwide.com	10,697
7	wc.yahoodns.net	8,235
8	fna.fbcdn.net	7,616
9	nrb.footprintdns.com	4,674
10	files.wordpress.com	4,655
11	u.fastly-insights.com	4,540
12	l4.adsco.re	4,029
13	services.video.ibm	3,978
14	aa.online-metrix.net	3,666
15	cdn.ampproject.org	3,660
Top effective second-level domains		
1	googlesyndication.com	452,598
2	office.com	186,072
3	cedexis-radar.net	49,663
4	gstatic.com	31,947
5	footprintdns.com	22,304
6	imrworldwide.com	15,448
7	cloudfront.net	10,394
8	adsco.re	10,032
9	fbcdn.net	9,160
10	akamaihd.net	8,653
11	yahoodns.net	8,240
12	fastly-insights.com	6,598
13	wordpress.com	6,418
14	amazonaws.com	5,783
15	sharepoint.com	5,257

Appendix B

Functional Specification of the Collectors and the Merger

This appendix provides a detailed specification of the collectors used in this thesis. The specification includes the input and output channels, the request and response models, the error codes, and the general behaviour of the collectors. At the beginning, there is a table listing the collector's inputs and outputs, referring to listings with the respective models. The specification is divided into sections, each describing a single collector. For general information on what the collectors do, refer to Section 6.4. Section B.10 shows an SQL representation of the data merger operation. Sections B.11, B.12, B.13 and B.14 provide the models used in the specification, examples of the input and output of the collectors, the collector response status codes, and the models and status codes used in the configuration mechanism, respectively.

B.1 Zone Collector

The input and output channels are defined in Table B.1. The collector has two side output channels that are used to provide requests for the DNS and RDAP-DN collectors.

Input		
Key:	`str`	a domain name
Value:	`ZoneRequest | None`	request options, see B.3
Main output (processed zone)		
Key:	`str`	the input domain name
Value:	`ZoneResult`	a collection result, see B.3
DNS requests output		
Key:	`str`	the input domain name
Value:	`DNSRequest`	a DNS collection request, see B.4
RDAP-DN requests output		
Key:	`str`	the input domain name
Value:	`RDAPDomainRequest`	a RDAP-DN collection request, see B.6

Table B.1: The zone collector's I/O paths.

Request: The request body is optional. If present, it contains two booleans that control whether the DNS and RDAP-DN requests will be published. It may also contain two sets that control the options of the generated DNS request. If the body is missing, the collector will publish both requests and will pass `None` values for the selts.

Response: A successful response contains a `ZoneData` value with:

- the zone domain name, the zone's SOA record, the public suffix,
- a set of all found secondary nameservers,
- a set of all resolved primary nameserver IPs,
- a set of all resolved secondary nameserver IPs,
- the `hasDNSKEY` field that will be `True` if a DNSKEY record exists, `False` if it does not, and `None` if any error occurred during the DNS query.

The three sets may be empty if the corresponding DNS resolutions fail.

If the zone was found, a DNS request and an RDAP-DN request are produced with the resulting zone information. If the request body contained DNS request options, they will be passed to the corresponding fields in the generated DNS request as-is.

Errors:

- `TIMEOUT` if a DNS response was not received in a configured time frame,
- `NOT_FOUND` if the SOA record was not found.

Remarks: When the input is a public suffix (e.g.`cz`, `co.uk` or `hakodate.hokkaido.jp`), the resolution is performed so that the result is the SOA record of the suffix. Otherwise, the public suffix is skipped (e.g. for `fit.vut.cz`, the query is made for `vut.cz` and `fit.vut.cz` but not `cz`). This is done to prevent skewing the results for names that do not exist in the DNS with legitimate values.

Example: See Listing B.12.

Input subscription: The zone collector typically subscribes to an external source of input domain names to process.

B.2 DNS Collector

The input and output channels are defined in Table B.2. The collector has two side output channels that are used to provide requests for the TLS collector and all the IP-based collectors.

Input		
Key:	`str`	a domain name
Value:	`DNSRequest | None`	request options, see B.4
Main output (DNS scan result)		
Key:	`str`	the input domain name
Value:	`DNSResult`	a collection result, see B.4
TLS requests output		
Key:	`str`	the input domain name
Value:	`str`	a selected target IP for the TLS collector
IP requests output		
Key:	`IPToProcess`	a pair of the input domain name and an IP address, see Section B.5
Value:	`None`	always empty

Table B.2: The DNS collector's I/O paths.

Request: The request body is required.

- The `zoneInfo` field must contain a valid zone data object (see B.3).
- The `typesToCollect` set is optional and controls which DNS record types will be queried. If the set is `None` or empty, a pre-configured value will be used.
- The `typesToProcessIPsFrom` set is optional and controls the source records types from which IP addresses will be published to the IP request output channel. If the set is `None`, a pre-configured value will be used. If the set is not `None` but empty, no IP addresses will be published.

Response: A successful response contains a `DNSData` value.

- Each property named by a record type will be not `None` iff the record exists in DNS and was fetched successfully; otherwise, it will be `None`.
- `ttlValues` is a map where the key is a successfully fetched record type and the value is the TTL value for the corresponding RRset.
- The `relatedIPs` properties of `CNAMERecord`, `MXRecord`, `NSRecord` may contain a set of IP addresses acquired by querying a common recursive DNS resolver for the A and AAAA records related to the CNAME value/MX value/nameserver.

IP requests are published for all IPs related to the records specified in the effective value of the `typesToProcessIPsFrom` set. A TLS request is generated if any IP address has been resolved:

1. as a related IP of a resolved CNAME record, or
2. from an A record, or
3. from an AAAA record.

A single address is selected from the records in this order. If an address related to a CNAME is selected, IPv4 is preferred. No other constraints are imposed on which address is selected.

Errors:

- `TIMEOUT` if **all** the issued DNS queries timed out (no response received in a configured time frame),
- `OTHER_DNS_ERROR` if **all** the issued DNS queries failed for another reason than timeout.
- In addition to the status and error fields, the data value bears information on per-query errors:
  - If an error occurs during a single DNS query, the corresponding property in the result is `None`. The `errors` map is populated with a pair keyed by the record type and a value giving a human-readable error description (e.g. "Timeout").
  - If all queries fail and at least one of the errors is not a timeout, the response will have the `OTHER_DNS_ERROR` status code but the `dnsData` field will have a value with the populated `errors` map.
  - If all queries fail with a timeout, the `dnsData` field will be `None`.

Remarks: The collector queries for the following record types and collects their values: A, AAAA, CNAME, MX, NS, TXT. IP addresses may be published to the respective side channel from the A, AAAA, CNAME, MX and NS records.

Example: See Listing B.13

Input subscription: The DNS collector typically subscribes to the "DNS request output" of the zone collector.

B.3 TLS Collector

The input and output channels are defined in Table B.3. The collector has no side output channels.

Input		
Key:	`str`	a domain name
Value:	`ip`	an IP address
Main output (TLS result)		
Key:	`str`	the input domain name
Value:	`TLSResult`	a collection result, see B.5

Table B.3: The TLS collector's I/O paths.

Request: The request body is required. It contains the IP address that the collector will connect to.

Response: A successful response contains a `TLSData` value.

- `fromIP` contains the IP address the collector connected to.

- `protocol` contains one of the values "TLSv1", "TLSv1.1", "TLSv1.2", "TLSv1.3", according to the chosen protocol.
- `cipher` contains an IANA name (description)[1] of the chosen ciphersuite.
- `certificates` contains a list of `Certificate` values. The list represents the certificate chain presented by the server and is order so that the leaf certificate comes first. Each value is a pair of the certificate Distinguished Name (DN) and its DER-encoded representation.

Errors:

- `UNSUPPORTED_ADDRESS` if the collector runs on a system that does not support the IP family of the input address (typically IPv6),
- `TIMEOUT` if the TLS handshake was not completed in a configured time frame (connection or socket I/O timed out),
- `CANNOT_FETCH` if the TLS handshake failed for another reason than timeout.

Remarks: The implementation must offer the TLS versions 1.0 to 1.3 to the remote hosts. The offered ciphersuites are not defined.

Example: See Listing B.14

Input subscription: The TLS collector typically subscribes to the "TLS request output" of the DNS collector.

B.4 RDAP-DN Collector

The input and output channels are defined in Table B.4. The collector has no side output channels.

Input		
Key:	`str`	a domain name
Value:	`RDAPDomainRequest	None`	request options, see B.6
Main output (RDAP-DN result)		
Key:	`str`	the input domain name
Value:	`RDAPDomainResult`	a collection result, see B.6

Table B.4: The RDAP-DN collector's I/O paths.

Request: The request body is optional. If present and the value of the `zone` field is not `None`, this value will be used as the RDAP (or WHOIS) query target.

[1]See `https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-4` for the list of known TLS ciphersuites.

Response: A response provides means to determine the success both of the RDAP query and the WHOIS query.

- `statusCode` signalises the result of the RDAP query.
- `rdapTarget` contains the target domain name that the query succeeded for (iff RDAP succeeds).
- `rdapData` contains the deserialised RDAP response JSON (iff RDAP succeeds).
- `entitites` contains the processed entities from the RDAP response (iff RDAP succeeds), see below.
- `whoisStatusCode` signalises the result (or lack thereof) of the WHOIS query.
- `whoisRaw` contains the raw WHOIS response (iff WHOIS succeeds). If several WHOIS requests were made (e.g. if the target WHOIS endpoint was determined by querying IANA's WHOIS service), this field is a list of raw responses ordered so that the response from the most specific endpoint comes first.

Errors (RDAP): The following codes may be returned in the `statusCode` field:

- `NO_ENDPOINT` if no RDAP endpoint is known for the TLD,
- `NOT_FOUND` if the RDAP endpoint returned HTTP status code 404 (not found, *i.e.* the DN does not exist at the queried RDAP endpoint),
- `RATE_LIMITED` if the RDAP endpoint returned HTTP status code 429 (too many requests),
- `TIMEOUT` if the RDAP query was not completed in a configured time frame,
- `CANNOT_FETCH` if the RDAP query failed for another reason (incl. another non-OK HTTP status code).
- `LOCAL_RATE_LIMIT`, `LRL_TIMEOUT` if the local rate limit is triggered, see below.

Errors (WHOIS): The following codes may be returned in the `whoisStatusCode` field:

- `WHOIS_NOT_PERFORMED` if the RDAP query succeeded, so no WHOIS query was made,
- all the error codes defined for the RDAP response may also be returned here with analogous meanings.

Entity processing: An RDAP-DN may contain a list of entities [56, Sec. 5.3]. An entity may specify, for example, information on the registrar and registrant. Some RDAP servers choose not to include the full details of an entity in the response, but rather provide a link (according to Section 4.2 of [56]) to the full entity data. If an entitites array is present in the RDAP response, the collector processes each entity:

- If the entity does not have a `links` key or has a `vcardArray` key, it is placed in the result `entities` field as-is.

- If the entity has a link with the `rel` attribute equal to “self”, an RDAP query is made for the target of this link.
- If the query succeeds, the `roles` array from the original entity is placed in the received object. Then, this modified entity object is placed in the result.
- If the query fails, the original entity object is placed in the result.

Rate limiting prevention: The collector must implement a local rate limiting mechanism to prevent triggering rate limiting at the remote RDAP servers. Queries to each individual RDAP endpoint are controlled by an independent, endpoint-specific local rate limiter. It is up to the implementation to decide how this mechanism is implemented – for example, the leaky bucket algorithm may be used. The parameters of the rate limiter must be configurable independently for each endpoint. Local rate limiting can be enforced in two modes:

- **Queueing:** The collector will queue the rate-limited requests and will retry them after the rate limit is lifted. Optionally, the collector may implement time bounding for the queued requests. In this case, the `LRL_TIMEOUT` error code is returned if the request is not processed within a configured time frame.
- **Immediate:** The collector will immediately return the `LOCAL_RATE_LIMIT` error code if the rate limit is triggered.

The immediate mode can be simulated by using the queueing mode with a very strict (low) time bound but the appropriate error code must always be used. The mode setting is a part of the collector’s configuration.

Example: See Listing B.15.

Input subscription: The RDAP-DN collector typically subscribes to the “RDAP-DN request output” of the zone collector.

B.5 Common Specification of the IP-Based Collectors

The key of messages accepted by the RDAP-IP, NERD, RTT and GEO-ASN collectors is always an `IPToProcess` value, a pair of a domain name and an IP address. Both IPv4 and IPv6 addresses are supported. The key model, as well as the request and response models, are defined in Listing B.1.

Requests: The request value may be `None` or an instance of `IPRequest`. If the request value or the `collectors` set is `None`, all collectors will process the request. If the `collectors` set is not `None` but empty, no collectors will process the request. Otherwise, a collector will only process the message if its type identifier is in the set.

Responses: The base result model for all IP-based collector results is `CommonIPResult`. It always has a field `collector` that contains the type identifier of the source collector. The `data` field is defined as a general object, each collector re-defines this field with a more specific type of the data it provides.

Errors: All collectors must validate the input address and return the `INVALID_ADDRESS` code if it is not.

```
class IPToProcess:
    dn: str
    ip: ip

class IPRequest:
    collectors: set[str] | None

class CommonIPResult(Result):
    collector: str
    # None iff statusCode != 0
    data: object | None
```

Listing B.1: The general IP models.

B.6 RDAP-IP Collector

The collector follows the common specification from Section B.5. The input and output channels are defined in Table B.5. The collector has no side output channels.

Input		
Key:	`IPToProcess`	a domain name/IP pair
Value:	`IPRequest | None`	request options, see above
Main output (RDAP-IP result)		
Key:	`IPToProcess`	the input pair
Value:	`RDAPIPResult`	a collection result, see B.7

Table B.5: The RDAP-IP collector's I/O paths.

Request: As described above for all IP-based collectors.

Response: The `data` field of a successful response contains the deserialised RDAP response JSON. It is taken as-is without any further processing.

Errors:

- `NOT_FOUND` if the RDAP endpoint returned HTTP status code 404 (not found, *i.e.* the IP does not exist),
- `RATE_LIMITED` if the RDAP endpoint returned HTTP status code 429 (too many requests),
- `TIMEOUT` if the RDAP query was not completed in a configured time frame,
- `CANNOT_FETCH` if the RDAP query failed for another reason (incl. another non-OK HTTP status code),

- `LOCAL_RATE_LIMIT`, `LRL_TIMEOUT` if the local rate limit is triggered, see RDAP-DN collector.

Rate limiting prevention: The collector must implement a local rate limiting mechanism to prevent triggering rate limiting at the remote RDAP servers. The mechanism is the same as described in RDAP-DN collector.

Example: See Listing B.16.

B.7 NERD Collector

The collector follows the common specification from Section B.5. The input and output channels are defined in Table B.6. The collector has no side output channels.

Input		
Key:	`IPToProcess`	a domain name/IP pair
Value:	`IPRequest | None`	request options, see above
Main output (NERD result)		
Key:	`IPToProcess`	the input pair
Value:	`NERDResult`	a collection result, see B.8

Table B.6: The NERD collector's I/O paths.

Request: As described above for all IP-based collectors.

Response: The `data` field of a successful response contains a `NERDData` value with a single field, `reputation`, that contains the retrieved reputation score. Note that NERD returns a zero score for unknown IPs. For IPv6 addresses, the result status code will always be `UNSUPPORTED_ADDRESS` as NERD does not support IPv6 at the moment[2].

Errors:

- `UNSUPPORTED_ADDRESS` if the input address is an IPv6 address,
- `TIMEOUT` if the NERD query was not completed in a configured time frame,
- `CANNOT_FETCH` if the NERD query failed for another reason (incl. a non-OK HTTP status code),
- `INVALID_RESPONSE` if the NERD response was not in the expected format (content length mismatch).

Example: See Listing B.17.

[2]While not namely unsupported, in time of writing, the implementation of the primary source for NERD, Warden, discards IPv6 addresses: `https://github.com/CESNET/NERD/blob/3b6972d/NERDd/warden_receiver.py#L500` (cit. 2024-07-04).

B.8 GEO-ASN Collector

The collector follows the common specification from Section B.5. The input and output channels are defined in Table B.7. The collector has no side output channels.

Input		
Key:	`IPToProcess`	a domain name/IP pair
Value:	`IPRequest | None`	request options, see above
Main output (GEO-ASN result)		
Key:	`IPToProcess`	the input pair
Value:	`GeoIPResult`	a collection result, see B.9

Table B.7: The GEO-ASN collector's I/O paths.

Request: As described above for all IP-based collectors.

Response: The `data` field of a successful response contains a `GeoIPData` value with data found in the GeoIP databases. If neither the City nor ASN database contains information on the IP, the status code will be `NOT_FOUND`. However, if the IP is found in only one of the databases, the response will be successful and the fields sourced from the database with missing data will be `None`.

Errors:

- `NOT_FOUND` if neither the City nor ASN database contains information on the IP.

Example: See Listing B.18.

B.9 RTT Collector

The collector follows the common specification from Section B.5. The input and output channels are defined in Table B.8. The collector has no side output channels.

Input		
Key:	`IPToProcess`	a domain name/IP pair
Value:	`IPRequest | None`	request options, see above
Main output (RTT result)		
Key:	`IPToProcess`	the input pair
Value:	`RTTResult`	a collection result, see B.10

Table B.8: The RTT collector's I/O paths.

Request: As described above for all IP-based collectors.

Response: The `data` field of a successful response contains a `RTTData` value with common information on the finished ping: the number of sent and received datagrams, the minimum, maximum and average RTT values, and the mean jitter (the mean absolute difference between consecutive RTTs), all in milliseconds.

Errors:

- `ICMP_DEST_UNREACHABLE` if a destination unreachable ICMP message was received.
- `ICMP_TIME_EXCEEDED` if a time exceeded ICMP message was received, *i.e.* a datagram was discarded due to the TTL field reaching zero.

Example: See Listing B.19.

B.10 The Data Merger Operation

Listing B.2 is a snippet of SQL code that describes the operation performed by the data merger. The relations ending with `Results` represent the most useful state stores, the actual messages are mapped to the relational model so that the key domain name is the `domainName` attribute; for IP-based collectors, there are also the `ip` and `collector` attributes. The complex message data object is stored in the `result` attribute. For the description of operation, refer to Section 6.5 and Figure 6.3.

```sql
-- Step 1: Aggregate collected IP data
WITH
-- The entries are essentialy mappings of unique
-- (DN, IP) -> map of (collector ID -> data)
AggregatedDataPerIP AS (
    SELECT
        domainName, ip,
        COLLECT_TO_MAP(collector,
            result->data) AS collectedData
    FROM IPCollectorResults
    GROUP BY domainName, ip),
-- The entries are essentially mappings of unique
-- DN -> map of (IP -> (map of collector ID -> data))
AggregatedDataPerDN AS (
    SELECT
        domainName,
        COLLECT_TO_MAP(ip, collectedData) AS ipDataMap
    FROM AggregatedDataPerIP
    GROUP BY domainName)
-- Step 2: Join all data by domain name
SELECT
    dns.domainName         domainName,
    zone.result->zone      zone,
    dns.result             dnsResult,
    tls.result             tlsResult,
    rdap.result            rdapDomainResult
    allIpResults.ipDataMap ipResults
FROM DNSResults dns
INNER JOIN ZoneResults zone
    ON dns.domainName = zone.domainName
INNER JOIN RDAPDomainResults rdap
    ON dns.domainName = rdap.domainName
LEFT JOIN TLSResults tls
    ON dns.domainName = tls.domainName
LEFT JOIN AggregatedDataPerDN allIpResults
    ON dns.domainName = allIpResults.domainName
-- Step 3: Only output complete data
WHERE dns.result IS NULL OR
    HAS_EXPECTED_IP_RESULTS(dns.result, allIpResults.ipDataMap) OR
    (HAS_IPS_FOR_TLS(dns.result) AND tls.domainName IS NOT NULL)
```

Listing B.2: The join operation perfomed by the merger.

B.11 Data Models

```
class ZoneRequest:
    collectDNS:   bool
    collectRDAP:  bool
    dnsTypesToCollect:        set[str] | None
    dnsTypesToProcessIPsFrom: set[str] | None

class ZoneResult(Result):
    # None iff statusCode != 0
    zone: ZoneInfo | None

class ZoneInfo:
    zone:         str
    soa:          SOARecord
    publicSuffix: str
    hasDNSKEY:    bool | None
    primaryNameserverIPs:     set[ip]
    secondaryNameservers:     set[str]
    secondaryNameserverIPs:   set[ip]

class SOARecord:
    primaryNS: str
    respMailboxDname: str
    serial:    str
    refresh:   int
    retry:     int
    expire:    int
    minTTL:    int
```

Listing B.3: Data models for the zone collector.

```
class DNSRequest:
    typesToCollect:        set[str] | None
    typesToProcessIPsFrom: set[ip]  | None
    zoneInfo:   ZoneInfo

class DNSResult(Result):
    # None iff statusCode not in (0, OTHER_DNS_ERROR)
    dnsData:    DNSData | None
    # None iff statusCode != 0
    ips:        list[IPFromRecord] | None

class IPFromRecord:
    ip:         ip
    rrType:     str

class DNSData:
    A:          set[ip] | None
    AAAA:       set[ip] | None
    CNAME:      CNAMERecord | None
    MX:         list[MXRecord] | None
    NS:         list[NSRecord] | None
    TXT:        list[str] | None
    # mappings of "A", "AAAA", ... -> error desc.
    errors:     dict[str, str] | None
    # mappings of "A", "AAAA", ... -> TTL value
    ttlValues:  dict[str, int]

class CNAMERecord:
    value:      str
    relatedIPs: set[ip] | None

class MXRecord:
    value:      str
    priority:   int
    relatedIPs: set[ip] | None

class NSRecord:
    nameserver: str
    relatedIPs: set[ip] | None
```

Listing B.4: Data models for the DNS collector.

```
class TLSResult(Result):
    # None iff statusCode != 0
    tlsData:  TLSData | None

class TLSData:
    fromIP:   ip
    protocol: str
    cipher:   str
    certificates: list[Certificate]

class Certificate:
    dn:       str
    derData:  bytes
```

Listing B.5: Data models for the TLS collector.

```
class RDAPDomainRequest:
    zone: str | None

class RDAPDomainResult(Result):
    rdapTarget:  str
    # both None iff statusCode != 0
    rdapData:    json | None
    entities:    json | None

    whoisStatusCode: int = -1
    # all None iff whoisStatusCode != 0
    whoisError:  str | None
    whoisRaw:    str | list[str] | None
```

Listing B.6: Data models for the RDAP-DN collector.

```
class RDAPIPResult(CommonIPResult):
    collector: str = "rdap-ip"
    data:      json | None
```

Listing B.7: Data models for the RDAP-IP collector.

```
class NERDResult(CommonIPResult):
    collector:  str = "nerd"
    data:       NERDData | None

class NERDData:
    reputation: float = 0.0
```

Listing B.8: Data models for the NERD collector.

```
class GeoIPResult(CommonIPResult):
    collector: str = "geo-asn"
    data:      GeoIPData | None

class GeoIPData:
    continentCode:  str | None
    countryCode:    str | None
    region:         str | None
    regionCode:     str | None
    city:           str | None
    postalCode:     str | None
    latitude:       float | None
    longitude:      float | None
    timezone:       str | None
    registeredCountryGeoNameId:  int | None
    representedCountryGeoNameId: int | None
    asn:            int | None
    asnOrg:         str | None
    networkAddress: str | None
    prefixLength:   int | None
```

Listing B.9: Data models for the GEO-ASN collector.

```
class RTTResult(CommonIPResult):
    collector: str = "rtt"
    data:      RTTData | None

class RTTData:
    min:       float
    avg:       float
    max:       float
    sent:      int
    received:  int
    jitter:    float
```

Listing B.10: Data models for the RTT collector.

```
class AllCollectedData:
    zone: ZoneInfo
    dnsResult: ExtendedDNSResult
    tlsResult: TLSResult | None
    rdapDomainResult: RDAPDomainResult | None
    ipResults: dict[str, dict[str, CommonIPResult]] | None
```

Listing B.11: The data model for the final gather operation.

B.12 Input/Output Examples

```
# Request:
("merlin.fit.vutbr.cz", None)

# Output (main channel):
# key = "merlin.fit.vutbr.cz"
ZoneResult(
    statusCode=0,
    error=None,
    lastAttempt="2024-06-14T20:00:00Z",
    zone=ZoneInfo(
        zone="fit.vutbr.cz",
        soaRecord=SOARecord(
            primaryNS="guta.fit.vutbr.cz",
            ...
        ),
        publicSuffix="cz",
        primaryNameserverIPs={"147.229.9.11",
            "2001:67c:1220:809::93e5:90b"},
        secondaryNameservers={"rhino.cis.vutbr.cz",
            "kazi.fit.vutbr.cz",
            "gate.feec.vutbr.cz"},
        secondaryNameserverIPs={...}
    )
)

# Output (DNS request):
# key = "merlin.fit.vutbr.cz"
DNSRequest(
    typesToCollect=None,
    typesToProcessIPsFrom=None,
    zoneInfo=ZoneInfo(
        zone="fit.vutbr.cz",
        ...
    )
)
```

```
# Output (RDAP-DN request):
# key = "merlin.fit.vutbr.cz"
RDAPDomainRequest(
    zone="fit.vutbr.cz"
)
```

Listing B.12: Example of a zone collector input and output.

```
# Request:
("fit.vut.cz", DNSRequest(
    typesToCollect={"A", "AAAA", "MX", "TXT"},
    typesToProcessIPsFrom={"A"},
    zoneInfo=ZoneInfo(
        zone="fit.vut.cz",
        ...
    )
))

# Output (main channel):
# key = "fit.vut.cz"
DNSResult(
    statusCode=0,
    error=None,
    lastAttempt="2024-06-14T20:00:00Z",
    dnsData=DNSData(
        A={"147.229.9.65", "1.2.3.4"},
        AAAA={"2001:67c:1220:809::93e5:941"},
        MX=[
            MXRecord(
                value="kazi.fit.vutbr.cz",
                priority=10,
                relatedIPs={"147.229.8.12",
                            "2001:67c:1220:808::93e5:80c"})
        ],
        TXT=[
            "v=spf1 a:kazi.fit.vutbr.cz ..."
        ],
        CNAME=None,
        NS=None,
        hasDNSKEY=True,
        errors=None,
        ttlValues={
            "A": 14400,
            "AAAA": 14400,
            "MX": 14400,
            "TXT": 14400
        }
    ),
    ips=[
        IPFromRecord(rrType="A",
                     ip="147.229.9.65"),
        IPFromRecord(rrType="A",
                     ip="1.2.3.4")
    ]
)

# Output (TLS request):
# key = "fit.vut.cz"
# value = "2001:67c:1220:809::93e5:941"

# Output (IP requests):
# two messages published, values are None, keys are:
IPToProcess(domainName="fit.vut.cz",
            ip="147.229.9.65"),
IPToProcess(domainName="fit.vut.cz",
            ip="1.2.3.4")
```

Listing B.13: Example of a DNS collector input and output.

```
# Request:
("fit.vut.cz", "147.229.8.12")

# Output (main channel):
# key = "fit.vut.cz"
TLSResult(
    statusCode=0,
    error=None,
    lastAttempt="2024-06-14T20:00:00Z",
    tlsData=TLSData(
        fromIP="147.229.8.12",
        protocol="TLSv1.3",
        cipher="TLS_AES_256_GCM_SHA384",
        certificates=[
            Certificate(dn="CN=imap.fit.vut.cz", derData=...),
            Certificate(dn="CN=R3, O=Let's Encrypt, C=US", derData=...),
        ]
    )
)
```

Listing B.14: Example of a TLS collector input and output.

```
# Request:
("fit.vut.cz", None)

# Output (main channel):
# key = "fit.vut.cz"
```

```
RDAPDomainResult(
    statusCode=0,
    error=None,
    lastAttempt="2024-06-14T20:00:00Z",
    rdapTarget="vut.cz",
    rdapData={
        "objectClassName": "domain",
                "rdapConformance": [
                        "rdap_level_0",
                        "fred_version_0"
                ],
        "handle": "vut.cz",
        ...
    },
    entities=[
        {
            "objectClassName": "entity",
                        "rdapConformance": [
                                "rdap_level_0"
                        ],
                        "handle": "SB:VUTBR-CZ",
                        "vcardArray": ...,
            ...
        }
    ],
    whoisStatusCode=WHOIS_NOT_PERFORMED,
    whoisError=None,
    whoisRaw=None
)

# Request:
("ondryaso.eu", None)

# Output (main channel):
# key = "ondryaso.eu"
RDAPDomainResult(
    statusCode=NO_ENDPOINT,
    error="No RDAP endpoint available for the domain name",
    lastAttempt="2024-06-14T20:00:00Z",
    rdapTarget="ondryaso.eu",
    rdapData=None,
    entities=None,
    whoisStatusCode=0,
    whoisError=None,
    whoisRaw="% The WHOIS service offered by EURid ..."
)
```

Listing B.15: Example of a RDAP-DN collector input and output.

```
# Request:
(IPToProcess(domainName="fit.vut.cz",
             ip="147.229.9.65"), None)

# Output (main channel):
# key = IPToProcess(...)
RDAPIPResult(
    statusCode=0,
    error=None,
    lastAttempt="2024-06-14T20:00:00Z",
    collector="rdap-ip",
    data={
        "handle": "147.229.0.0 - 147.229.254.255",
        "name": "VUTBRNET",
        "ipVersion": "v4",
        "country": "CZ",
        "entities": [...],
        ...
    }
)
```

Listing B.16: Example of a RDAP-IP collector input and output.

```
# Request:
(IPToProcess(domainName="fit.vut.cz",
             ip="147.229.9.65"), None)

# Output (main channel):
# key = IPToProcess(...)
NERDResult(
    statusCode=0,
    error=None,
    lastAttempt="2024-06-14T20:00:00Z",
    collector="nerd",
    data=NERDData(
        reputation=0.0
    )
)
```

Listing B.17: Example of a NERD collector input and output.

```
# Request:
(IPToProcess(domainName="fit.vut.cz",
             ip="147.229.9.65"), None)

# Output (main channel):
# key = IPToProcess(...)
GeoIPResult(
    statusCode=0,
    error=None,
    lastAttempt="2024-06-14T20:00:00Z",
    collector="geo-asn",
    data=GeoIPData(
        continentCode="EU",
        countryCode="CZ",
        latitude=49.2067,
        longitude=16.5888,
        ...,
        asn=197451,
        asnOrg="Brno University of Technology",
        networkAddress="147.229.0.0",
        prefixLength=17
    )
)
```

Listing B.18: Example of a GEO-ASN collector input and output.

```
# Request:
(IPToProcess(domainName="fit.vut.cz",
             ip="147.229.9.65"), None)

# Output (main channel):
# key = IPToProcess(...)
RTTResult(
    statusCode=0,
    error=None,
    lastAttempt="2024-06-14T20:00:00Z",
    collector="rtt",
    data=RTTData(
        min=0.812,
        avg=1.099,
        max=1.657,
        sent=5,
        received=5,
        jitter=0.446
    )
)
```

Listing B.19: Example of a RTT collector input and output.

B.13 Collector Result Codes

Code	Name	Description
Success and general errors		
0	`SUCCESS`	The collection was successful.
1	`INVALID_MESSAGE`	Invalid input message format (model deserialisation error).
2	`INVALID_DOMAIN_NAME`	Invalid domain name in request.
3	`INVALID_ADDRESS`	Invalid IP address in request.
4	`UNSUPPORTED_ADDRESS`	The IP address is valid but the collector cannot process it (e.g., IPv6 is not available).
5	`INTERNAL_ERROR`	Unspecified internal error.
General remote fetch errors		
10	`CANNOT_FETCH`	General error when fetching data (e.g., non-success status code in the response from the remote party).
11	`TIMEOUT`	Could not finish the request in a configured time.
12	`NOT_FOUND`	No data found at the remote party for the requested domain name/IP address.
13	`RATE_LIMITED`	The remote party rate-limited the request.
14	`INVALID_RESPONSE`	Invalid format of the response from the remote party (deserialisation error).
15	`LOCAL_RATE_LIMIT`	Local rate limiter in the immediate mode prevented the request.
16	`LRL_TIMEOUT`	Could not pass the local rate limiter in a configured time.
DNS-specific errors		
20	`OTHER_DNS_ERROR`	of all issued queries (for all RRtypes) failed. `dnsData` is not null and its `errors` field is set.
RDAP-DN-specific errors		
30	`NO_ENDPOINT`	No RDAP endpoint found for the domain (TLD).
35	`WHOIS_NOT_PERFORMED`	RDAP succeeded, no WHOIS query was made.
RTT-specific errors		
40	`ICMP_DEST_UNREACHABLE`	The remote host or its inbound gateway indicated that the destination is unreachable for some reason.
41	`ICMP_TIME_EXCEEDED`	The datagram was discarded due to the TTL field reaching zero.

B.14 Collector Configuration

```
class ConfigurationValidationError:
    propertyPath: str
    errorCode:    int
    error:        str | None
    soft:         bool

class ConfigurationChangeResult:
    success: bool
    errors:  list[ConfigurationValidationError] | None
    message: str | None
    currentConfig: dict[str, json]
```

Listing B.20: Data model for a configuration change result message.

Code	Name	Description
1	`OTHER`	Unspecified error.
2	`INVALID_MESSAGE`	Invalid input message format (configuration model deserialisation error).
3	`INVALID_PROPERTY`	No such configuration property exists.
4	`INVALID_TYPE`	Invalid data type of the provided value.
5	`OUT_OF_RANGE`	The provided value is out of the allowed range.
6	`READ_ONLY`	The property cannot be changed dynamically.
7	`MISSING`	The property must be explicitly defined.

Table B.9: The configuration validation error codes.

Appendix C

Component Identifiers

Each pipeline component is assigned a component ID. It is used in the configuration exchange keys and as the name of the Docker Compose service. Additionally, the collector components have a collector ID, used inside the messages (e.g. for specifying which IP collector should run). One of the identifiers is used in the generation of Kafka consumer group ID, Kafka Streams application ID, and Faust application ID.

Component	**Component ID**	**Collector ID**
Zone collector	`collector-zone`	`zone`
DNS collector	`collector-dns`	`dns`
TLS collector	`collector-tls`	`tls`
RDAP-DN collector	`collector-rdap-dn`	`rdap-dn`
RDAP-IP collector	`collector-rdap-ip`	`rdap-ip`
NERD collector	`collector-nerd`	`nerd`
GEO-ASN collector	`collector-geoip`	`geo-asn`
RTT collector	`collector-rtt`	`rtt`
Data merger	`merger`	-
Feature extractor	`extractor`	-
Classifier unit	`classifier-unit`	-
Loader & Pre-filter	`loader`	-

Appendix D

Examples of Collector Implementation

```
from common import read_config, make_app, ensure_model, log
COLLECTOR = "Faust app ID (Kafka group ID)"
COMPONENT_NAME = "collector-" + COLLECTOR
# 1. Logging and configuration
config = read_config()
component_config = config.get(COLLECTOR, {})
logger = log.init(COMPONENT_NAME, config)
CONCURRENCY = component_config.get("concurrency", 4)
# (read other configuration values)
# 2. Init the Faust application
app = make_app(COLLECTOR, config)
# 3. Define the used topics (additional side outputs may be used)
topic_to_process = app.topic("to_process_X", key_type=str)
topic_processed = app.topic("processed_X")
# 4. Custom functions
# 5. The Faust agent processing the input topic
@app.agent(topic_to_process, concurrency=CONCURRENCY)
async def process_entries(stream):
  # 6. Custom initialization
  # 7. The async for loop consumes incoming events
  async for dn, value in stream.items():
    # 8. Deserialise the request
    #    ensure_model never throws but may return None
    request = ensure_model(RequestModelClass, value)
    try:
      # 9. Custom processing
      response = await custom_processing_fun(dn, request)
      # 10. Produce the result(s)
      await topic_processed.send(key=dn, value=response)
    except Exception as e:
      # Main logging
      logger.k_unhandled_error(e, dn)
      # Produces a result with the INTERNAL_ERROR status code
      await handle_top_level_component_exception(e, COLLECTOR, dn,
        ResultModelClass, topic_processed)  # from collectors.utils
```

Listing D.1: A Faust-based collector module framework.

```python
from common import read_config, make_app, ensure_model, log
COLLECTOR = "Faust app ID (Kafka group ID)"
COMPONENT_NAME = "collector-" + COLLECTOR
# 1. Logging and configuration
config = read_config()
component_config = config.get(COLLECTOR, {})
logger = log.init(COMPONENT_NAME, config)
CONCURRENCY = component_config.get("concurrency", 4)
# (read other configuration values)
# 2. Init the Faust application
app = make_app(COLLECTOR, config)
# 3. Define the used topics (additional side outputs may be used)
topic_to_process = app.topic("to_process_IP")
topic_processed = app.topic("collected_IP_data")
# 4. Custom functions
# 5. The Faust agent processing the input topic
@app.agent(topic_to_process, concurrency=CONCURRENCY)
async def process_entries(stream):
  # 6. Custom initialization
  # 7. The async for loop consumes incoming events
  async for dn_ip, value in stream.items():
    # 8. Deserialise the domain name / IP pair and the request
    #    ensure_model never throws but may return None
    dn_ip = ensure_model(IPToProcess, dn_ip)
    if dn_ip is None:
      continue  # Deserialisation error
    request = ensure_model(IPProcessRequest, value)
    # 9. Omit the entry if the collector is not requested
    if request is not None and request.collectors is not None and \
      COLLECTOR not in request.collectors:
      continue
    try:
      # 10. Custom processing
      response = await custom_processing_fun(dn_ip, request)
      # 11. Produce the result(s)
      await topic_processed.send(key=dn_ip, value=response)
    except Exception as e:
      # Main logging
      logger.k_unhandled_error(e, str(dn_ip))
      # Produces a result with the INTERNAL_ERROR status code
      await handle_top_level_component_exception(e, COLLECTOR, dn_ip,
        ResultModelClass, topic_processed)  # from collectors.utils
```

Listing D.2: A Faust-based IP collector module framework.

```java
public class SomeCollector
  extends BaseStandaloneCollector<String, InValueType> {
  // Collector ID and component ID
  public static final String NAME = "some";
  public static final String COMPONENT_NAME = "collector-" + NAME;
  // Logging
  private static final org.slf4j.Logger Logger =
   Common.getComponentLogger(SomeCollector.class);
  // Options
  private final int _batchSize, _timeout;
  // Producer(s)
  private final KafkaProducer<String, ResultType> _mainProducer;
  // Async task executor
  private final ExecutorService _executor;

  public Collector(@NotNull ObjectMapper jsonMapper,
                   @NotNull String appName,
                   @NotNull Properties properties) {
    super(jsonMapper, appName, properties,
      // The base constructor expects a Serde (serializer/deserializer)
      // for the input key and value.
      Serdes.String(), JsonSerde.of(jsonMapper, InValueType.class));
    // Initialize the options
    _batchSize = Integer.parseInt(properties.getProperty(
      "collectors.some.batch.size", "32"));
    _timeout = Integer.parseInt(properties.getProperty(
      "collectors.some.timeout", "1000"));
    // Initialize the producer(s) using the given serializers
    _mainProducer = super.createProducer(new StringSerializer(),
      new JsonSerializer<ResultType>(jsonMapper));
    // Initialize the async task executor
    _executor = Executors.newVirtualThreadPerTaskExecutor();
  }

  @Override
  public void run(CommandLine cmd) {
    // Initialize the parallel processor
    super.buildProcessor(_batchSize);
    // Subscribe to the source topic
    _parallelProcessor.subscribe(UniLists.of(Topics.IN_TOPIC));
    // Start the parallel processor
    _parallelProcessor.poll(entryContext -> {
      final String key = entryContext.key();
      final InValueType value = entryContext.value();
      // Start the processing
      var resultFuture = customProcess(key, value)
        .orTimeout(_timeout, TimeUnit.MILLISECONDS);
      // Wait for the result on the processor's thread
```

```
        try {
          resultFuture.join();
        } catch (CompletionException e) {
          if (e.getCause() instanceof TimeoutException) {
            // The processing timeout was triggered
            _mainProducer.send(new ProducerRecord<>(Topics.OUT_TOPIC, key,
              errorResult(ResultCodes.TIMEOUT, "Timeout")));
          } else {
            // The processing failed with an exception
            _mainProducer.send(new ProducerRecord<>(Topics.OUT_TOPIC, key,
              errorResult(ResultCodes.INTERNAL_ERROR, e.getMessage())));
          }
        }
      });
    }

    private CompletableFuture<Void> customProcess(String key,
                                                  InValueType value) {
      // Schedule the processing on the executor
      return CompletableFuture.runAsync(() -> {
        // Process the input somehow
        final ResultType result = ...;
        // Send the result
        _mainProducer.send(new ProducerRecord<>(Topics.OUT_TOPIC, key,
          result));
      }, _executor);
    }
  }
```

Listing D.3: A generalised structure of a collector based on Confluent Parallel Consumer.

Appendix E

Examples of Configuration Mappings

```
{
  "connection": {
    "ssl": {
      "ca_file": "evil.pem"
    }
  },
  "faust": {
    "blocking_timeout": 10.5,
    "something": null
  },
  "rdap_ip": {
    "app_id": "this-wont-be-used",
    "http_timeout_sec": 5
  }
}
```

(a) The exchange object

```
#
# kept from the previous config:
[connection.ssl]
ca_file="legit_ca.pem"
#
#
[faust]
blocking_timeout=10.5
# null value omitted
#
[rdap_ip]
app_id="rdap-ip-collector"
http_timeout_sec=5
#
#
```

(b) The target properties file

Listing E.1: Example of the configuration mapping for Python collectors.

```
{
  "collector": {
    "max.concurrency": 32,
    "nerd.token": "xyz"
  },
  "system": {
    "compression.type": "zstd",
    "security.foobar": "baz"
  }
}
```

(a) the exchange object

```
#
#
collectors.max.concurrency=32
collectors.nerd.token=xyz
#
#
compression.type=zstd
# security.foobar is not mapped
# kept from the previous config:
security.protocol=SSL
```

(b) the target properties file

Listing E.2: Example of the configuration mapping for Java collectors.

Appendix F

List of Features

Each feature is prefixed with a specifier of the feature's category noted in the heading of the respective part of the table. The quoted implementation files can be found in `src/python_pipeline/extractor/extractor/transformations`. Features with references were adopted from related literature, those without references were proposed by the DomainRadar team. Curly braces in the feature name column denote that the several features are actually computed.

Feature name	**Description and references**
	Lexical features (`lex_` prefix, impl. in `lexical.py`)
`name_len`	Length of the domain name [114, 53, 49, 6]
`has_digit`	True if the DN contains a digit [41]
`phishing_keyword_count`	Occurence count of 45 phishing keywords [41]
`benign_keyword_count`	Occurence count of 43 benign keywords [41]
`consecutive_chars`	Longest consecutive sequence length [114, 53, 49]
`tld_len`	Length of the TLD
`tld_abuse_score`	Score for most-abused TLD according to [1]
`tld_hash`	Hash of the TLD
`sld_len`	Length of the second-level domain (SLD)
`sld_norm_entropy`	Normalised entropy of the SLD
`sld_phishing_keyword_count`	Occurence count of 47 phishing keywords in the SLD
`sub_count`	Number of subdomains (domain level) [99]
`stld_unique_char_cnt`	Number of unique characters in the TLD and SLD
`begins_with_digit`	True if the name begins with a digit
`www_flag`	True if the name begins with "www"
`sub_max_conson_len`	Longest consonant sequence length in subdomains [41]
`sub_norm_entropy`	Normalised entropy of subdomains [114, 49, 42, 113]
`{sub,sld}_digit_count`	Number of digits in the subdomains/SLD [99]
`{sub,sld}_digit_ratio`	Ratio of digits in the subdomains/SLD
`{sub,sld}_vowel_count`	Number of vowels in the subdomains/SLD[113]
`{sub,sld}_vowel_ratio`	Ratio of vowels in the subdomains/SLD
`{sub,sld}_consonant_count`	Number of consonants in the subdomains/SLD
`{sub,sld}_consonant_ratio`	Ratio of consonants in the subdomains/SLD
`{sub,sld}_non_alpanum_count`	Total number of hyphens in the subdomains/SLD[99]
`{sub,sld}_non_alpanum_ratio`	Ratio of underscores and hyphens in the subdomains/SLD
`{sub,sld}_hex_count`	Number of hex symbols in the subdomains/SLD
`{sub,sld}_hex_ratio`	Ratio of hex symbols in the subdomains/SLD
`{phishing, malware, dga}_ bigram_matches`	Number of common phishing/malware/DGA 2-gram matches [125]
`{phishing, malware, dga}_ trigram_matches`	Number of common phishing/malware/DGA 3-gram matches [125]

Feature name	**Description and references**
`{phishing, malware, dga}_tetragram_matches`	Number of common phishing/malware/DGA 4-gram matches [125]
`{phishing, malware, dga}_pentagram_matches`	Number of common phishing/malware/DGA 5-gram matches [125]
`avg_part_len`	Average length of domain name parts
`stdev_part_lens`	Standard deviation of domain name part lengths
`longest_part_len`	Length of the longest domain name part
`shortest_sub_len`	Length of the shortest subdomain
`ipv4_in_domain`	True if the DN contains an IPv4 address
`has_{trusted, wellknown, cdn, vps, img}_suffix`	True if the DN ends with one of well-known suffixes
`suffix_score`	$10T + 5W + 3C + 2V + 8I$ where $T = 1 \Leftrightarrow$ `has_trusted_suffix` is true, etc.

DNS-based features (`dns_` prefix, impl. in `dns.py`)	
`A_count`	Number of A records [100]
`AAAA_count`	Number of AAAA records
`MX_count`	Number of MX records [70, 105]
`NS_count`	Number of NS records [70]
`TXT_count`	Number of TXT records
`CNAME_count`	Number of CNAME records
`resolved_record_types`	Number of discovered RRsets
`has_dnskey`	True if a DNSKEY RRset was found in the zone
`dnssec_score`	DNSSEC scoring (always zero, for compatibility only)
`ttl_avg`	Average TTL value across RRsets [49, 53, 100, 101, 114]
`ttl_stdev`	Standard deviation of TTLs across RRsets [49, 53, 114]
`ttl_low`	Number of RRsets with TTL $\in$ [0,100] [17]
`ttl_mid`	Number of RRsets with TTL $\in$ [101,500] [17]
`ttl_distinct_count`	Number of distinct TTL values across RRsets [17]
`soa_refresh`	SOA refresh parameter
`soa_retry`	SOA retry parameter
`soa_expire`	SOA expire parameter
`soa_min_ttl`	SOA minimum TTL
`domain_name_in_mx`	True if any mailserver is a subdomain of the DN
`txt_external_verification_score`	Number of known vendor verification strings in TXT RRs (e.g. `google-site-verification=`)
`txt_spf_exists`	True if an SPF record is in the TXT RRs
`txt_dkim_exists`	True if a DKIM record is in the TXT RRs
`txt_dmarc_exists`	True if a DMARC record is in the TXT RRs
DNS-based lexical features	
`zone_level`	Number of subdomains in the zone DN
`zone_digit_count`	Number of digits in the zone DN
`zone_len`	Number of characters in the zone DN
`zone_entropy`	Normalised entropy of the zone DN
`soa_primary_ns_level`	Number of subdomains in the primary NS DN
`soa_primary_ns_digit_count`	Number of digits in the primary NS DN
`soa_primary_ns_len`	Number of characters in the primary NS DN

Feature name	**Description and references**
`soa_primary_ns_entropy`	Normalised entropy of the primary NS DN
`soa_email_level`	Number of subdomains in the admin's email DN
`soa_email_digit_count`	Number of digits in the admin's email DN
`soa_email_len`	Number of characters in the admin's email DN
`soa_email_entropy`	Normalised entropy of the admin's email DN
`mx_avg_len`	Average number of characters of the DNs in MX records
`mx_avg_entropy`	Average Normalised entropy of the DNs in MX records
`txt_avg_len`	Average length of TXT RRs values
`txt_avg_entropy`	Average normalised entropy of TXT RRs values
IP-based features (`ip_` prefix, impl. in `ip.py`)	
`count`	Number of IP addresses [114, 17, 53, 49, 6, 4]
`mean_average_rtt`	Average RTT of all ICMP Echo attempts
`v4_ratio`	Ratio: IPv4 count : all related IPs count
`aaaa_to_all_ratio`	Ratio: A/AAAA-sourced IPs count : all related IPs count
`entropy`	Entropy of /16 IPv4 prefixes + entropy of /64 IPv6 prefixes [101, 51]
`as_address_entropy`	Entropy of autonomous system (AS) IP prefixes [51]
`asn_entropy`	Entropy of AS numbers (ASNs) [99, 113]
`distinct_as_count`	Number of distinct ASNs [5, 6, 100]
Features based on domain registration data (`rdap_` prefix, impl. in `rdap_dn.py`)	
`registration_period`	Diff. between expiration and registration date [114, 53, 49]
`domain_age`	Days elapsed since the domain registration [100]
`time_from_last_change`	Days elapsed since the last change [113]
`domain_active_time`	min(today, expiration) - registration date [114, 53, 49]
`has_dnssec`	True if domain uses DNSSEC (according to RDAP)
`registrar_name_len`	Length of the registrar's name [99, 113, 100]
`registrar_name_entropy`	Entropy of the registrar's name [99, 113, 100]
`registrar_name_hash`	Hash of the registrar's name [99, 113, 100]
`registrant_name_len`	Length of the registrant's name [99, 113]
`registrant_name_entropy`	Entropy of the registrant's name [99, 113]
`admin_name_len`	Length of the administrative contact's name
`admin_name_entropy`	Entropy of the administrative contact's name
`admin_email_len`	Length of the administrative contact's email [70]
`admin_email_entropy`	Entropy of the administrative contact's email [70]
Features based on IP registration data (`rdap_ip_` prefix, impl. in `rdap_ip.py`)	
`v4_count`	Number of IPv4 addresses with avail. RDAP data
`v6_count`	Number of IPv6 addresses with avail. RDAP data
`shortest_v4_prefix_len`	Length of the shortest IPv4 prefix
`longest_v4_prefix_len`	Length of the longest IPv4 prefix
`shortest_v6_prefix_len`	Length of the shortest IPv6 prefix
`longest_v6_prefix_len`	Length of the longest IPv6 prefix
`avg_admin_name_len`	Average length of the admins' names
`avg_admin_name_ent`	Average entropy of the admins' names
`avg_admin_email_len`	Average length of the admins' emails
`avg_admin_email_ent`	Average entropy of the admins' emails

Feature name	**Description and references**
	TLS-based features (`tls_` prefix, impl. in `tls.py`)
`has_tls`	True if a TLS conn. to 443 was established [115, 90]
`chain_len`	Length of the certificate chain [4]
`is_self_signed`	True if leaf ceriticate is self-signed [4, 121]
`root_authority_hash`	Hash of the root certificate's authority name
`leaf_authority_hash`	Hash of the leaf certificate's authority name
`negotiated_version_id`	Negotiated TLS version number (TLSv1.x)
`negotiated_cipher_id`	Identifier of the negotiated TLS cipher [16, 4]
`root_cert_validity_len`	Length of the validity period of the root certificate
`leaf_cert_validity_len`	Length of the validity period of the leaf cert. [121, 90, 4]
`broken_chain`	True if there is a certificate that was never valid
`expired_chain`	True if there is an expired certificate in the chain
`total_extension_count`	Total extensions in all certificates in the chain [16, 121]
`critical_extensions`	Total extensions flagged as "critical" in all certificates
`with_policies_crt_count`	Number of certificates that include the "policies" extension
`percentage_crt_ with_policies`	Ratio: with the "policies" extension : all
`x509_anypolicy_crt_count`	Number of certificates not enforcing any policy
`iso_policy_crt_count`	Total discovered policies in the 1.* OID space
`joint_isoitu_policy_ crt_count`	Total discovered policies in the 2.* OID space
`subject_count`	Number of subject alt. names (SANs) in the leaf cert. [121, 4]
`server_auth_crt_count`	Number of certs. with "Server Authentication" key usage
`client_auth_crt_count`	Number of certs. with "Client Authentication" key usage
`unique_SLD_count`	Number of unique SAN of the "DNS name" type
`CA_certs_in_chain_ratio`	Ratio: CA certificates : all
`common_name_count`	Number of common names in the chain
	Geolocation-based features (`geo_` prefix, impl. in `geo.py`)
`countries_count`	Number of distinct countries [114, 17, 53, 49, 6]
`continent_count`	Number of distinct continents
`countries_hash`	Unique hash for each combination of countries [99]
`continent_hash`	Unique hash for each combination of continents
`malic_host_country`	Number of IPs from specific countries
`lat_stdev`	Standard deviation of the IP location latitudes
`lon_stdev`	Standard deviation of the IP location longitudes
`{min, max, mean}_lat`	Minimum/maximum latitude and the average of all lat.
`{min, max, mean}_lon`	Minimum/maximum longitude and the average of all long.
`lat_range`	`max_lat` − `min_lat`
`lon_range`	`max_lon` − `min_lon`
`centroid_lat`	(`max_lat` + `min_lat`)/2
`centroid_lon`	(`max_lon` + `min_lon`)/2
`estimated_area`	`lat_range` × `lon_range`

Appendix G

Used Packages and Licences

The following table lists the libraries used in the project, including transitive dependencies, and their licences. The table uses these abbreviations:

Apache Apache License, version 2.0:
https://www.apache.org/licenses/LICENSE-2.0

BSD The 3-Clause BSD License:
https://opensource.org/license/bsd-3-clause

BSD2 The 2-Clause BSD License:
https://opensource.org/license/bsd-2-clause

CC0 Creative Commons Zero 1.0 Universal:
https://creativecommons.org/public-domain/cc0/

EPL 1.0/2.0 Eclipse Public License:
Version 1.0: https://www.eclipse.org/legal/epl-v10.html
Version 2.0: https://www.eclipse.org/legal/epl-2.0/

ISC ISC License:
https://opensource.org/license/isc-license-txt

LGPL GNU Lesser General Public License, version 3:
https://www.gnu.org/licenses/lgpl-3.0.en.html

MIT The MIT License:
https://opensource.org/license/mit

Repoze Repoze Public License (a derivative of the BSD License):
https://github.com/Pylons/venusian/blob/main/LICENSE.txt

The Unlicense https://choosealicense.com/licenses/unlicense/

Package	**Version**	**Licence**
	Python	
aiohttp	3.9.5	Apache
aiohttp_cors	0.7.0	Apache
aiokafka	0.10.0	Apache
annotated-types	0.7.0	MIT
async-timeout	4.0.3	Apache
asynciolimiter	1.0.0	MIT

Package	**Version**	**Licence**
asyncwhois	1.1.4	MIT
cffi	1.16.0	MIT
click	8.1.7	BSD
cramjam	2.8.3	MIT
croniter	2.0.5	MIT
cryptography	42.0.8	Apache or BSD
dnspython	2.6.1	ISC
faust-streaming	0.11.0	BSD
feather-format	0.4.1	Apache
filelock	3.15.4	The Unlicense
fsspec	2024.6.1	Apache
httpx	0.27.0	BSD
icmplib	3.0.4	LGPL
idna	3.7	BSD
intervaltree	3.1.0	Apache
mode-streaming	0.4.1	BSD
mypy-extensions	1.0.0	MIT
numpy	2.0.0	BSD
opentracing	2.4.0	Apache
pandas	2.2.2	BSD
pyarrow	16.1.0	Apache
pydantic	2.8.2	MIT
pydantic-core	2.20.1	MIT
python-dateutil	2.9.0	Apache or BSD
python-socks	2.5.0	Apache
python-whois	0.9.4	MIT
requests	2.32.3	Apache
requests-file	2.1.0	Apache
six	1.16.0	MIT
terminaltables	3.1.10	MIT
tldextract	5.1.2	BSD
tzdata	2024.1	Apache
venusian	3.1.0	Repoze
whodap	0.1.12	MIT
whoisit	2.7.7	BSD
yarl	1.9.4	Apache

Package	Version	Licence
Java		
Jackson-annotations	2.17.1	Apache
Jackson-core	2.17.1	Apache
Jackson-databind	2.17.1	Apache
Jackson datatype: JSR310	2.17.1	Apache
zstd-jni	1.5.5	BSD2
FindBugs-jsr305	3.0.2	Apache
error-prone annotations	2.26.1	Apache
Guava: Google Core Libraries for Java	33.1.0	Apache
Guava and InternalFutures	1.0.2	Apache
Guava ListenableFuture only	9999.0	Apache
J2ObjC Annotations	3.0.0	Apache
MaxMind DB Reader	3.1.0	Apache
MaxMind GeoIP2 API	4.2.0	Apache
Apache Commons CLI	1.6.0	Apache
Confluent Parallel Consumer Core	0.5.3.0	Apache
micrometer-commons	1.13.0	Apache
micrometer-core	1.13.0	Apache
micrometer-observation	1.13.0	Apache
javax.ws.rs-api	2.1.1	EPL 2.0
JUnit	4.13.1	EPL 1.0
Apache Kafka	3.7.0	Apache
(Clients, Streams, Connect API)		
Checker Qual	3.42.0	MIT
Hamcrest Core	1.3	BSD
HdrHistogram	2.2.1	BSD2
JetBrains Java Annotations	24.1.0	Apache
LatencyUtils	2.0.3	CC0
LZ4 and xxHash	1.8.0	Apache
RocksDB JNI	7.9.2	Apache
SLF4J API Module	2.0.13	MIT
SLF4J Simple Provider	2.0.13	MIT
snappy-java	1.1.10.5	Apache
UniJ	0.1.3	Apache

Appendix H

Results of the Evaluation Experiments

For the explanation of the following tables and charts, refer to Chapter 8. Space is used as the digit group separator in large numbers to prevent confusion with the decimal point.

H.1 Collector Experiments

Note that the RTT collector never exhibited errors so it is omitted from the error tables and charts for clarity. The GEO-ASN collector is omitted from the charts because the error rate was always below 0.1%. The categories used in the error rate tables map to the collector result codes as follows:

- NotFound: `NOT_FOUND`, `UNSUPPORTED_ADDRESS`,
- Remote: `CANNOT_FETCH`, `TIMEOUT`, `OTHER_DNS_ERROR`,
- Internal: `INTERNAL_ERROR`,
- RateLimit: `RATE_LIMITED`,
- NoEndpt: `NO_ENDPOINT`,
- Other: all other non-zero result codes.

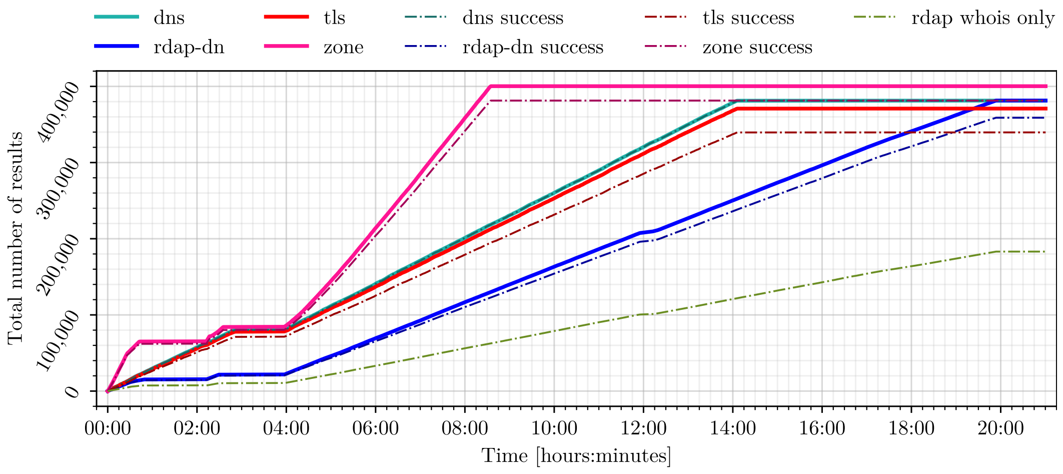
Experiment #1 (2 partitions)

Collector	Tot [h:m]	AvgTput [req/s]	AvgColT [ms]	MnQdColT [s]
Zone	8:34	12.96	77	18 704
DNS	14:06	7.51	133	7 862
TLS	14:06	7.30	137	0.370
RDAP-DN	19:53	5.33	188	21 245
GEO-ASN	14:06	18.12	55	0.008
NERD	14:06	18.15	55	0.018
RDAP-IP	19:02	13.42	75	10 066
RTT	21:00*	0.43	2 314	36 289

Table H.1: Per-collector metrics in experiment #1. The RTT collector was extremely slow and was terminated prematurely.

		Zone	DNS	TLS	R-DN	*WHOIS*	GEO	NERD	R-IP
NotFound	req%	4.20	0	0	0.12	0.46	0.03	29.75	0
	all%	4.21	0	0	0.11	-	-	-	-
Remote	req%	0.14	0.03	8.44	15.27	0	0	0.65	0.02
	all%	0.14	0.03	7.82	14.56	-	-	-	-
Internal	req%	0.38	0.03	0	1.91	10.49	0	0	0.02
	all%	0.38	0.03	0	1.82	-	-	-	-
RateLimit	req%	-	-	-	11.39	0	-	-	0
	all%	-	-	-	10.86	-	-	-	-
NoEndpt	req%	-	-	-	19.57	0	-	-	0
	all%	-	-	-	18.66	-	-	-	-
Other	req%	0	0	0	5.66	0	0	0	0
	all%	0	0	0	5.39	-	-	-	-
Total	req%	4.72	0.06	8.44	53.92	10.95	0.03	30.40	0.04
	all%	4.72	0.05	7.82	51.41	0	0	0	0

Table H.2: Error rates in experiment #1.



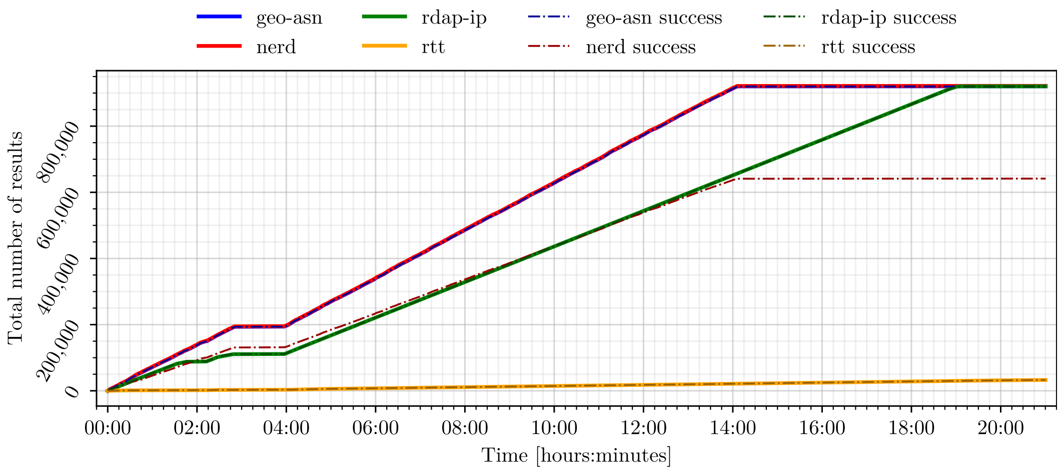


Figure H.1: Number of responses over time in experiment #1.

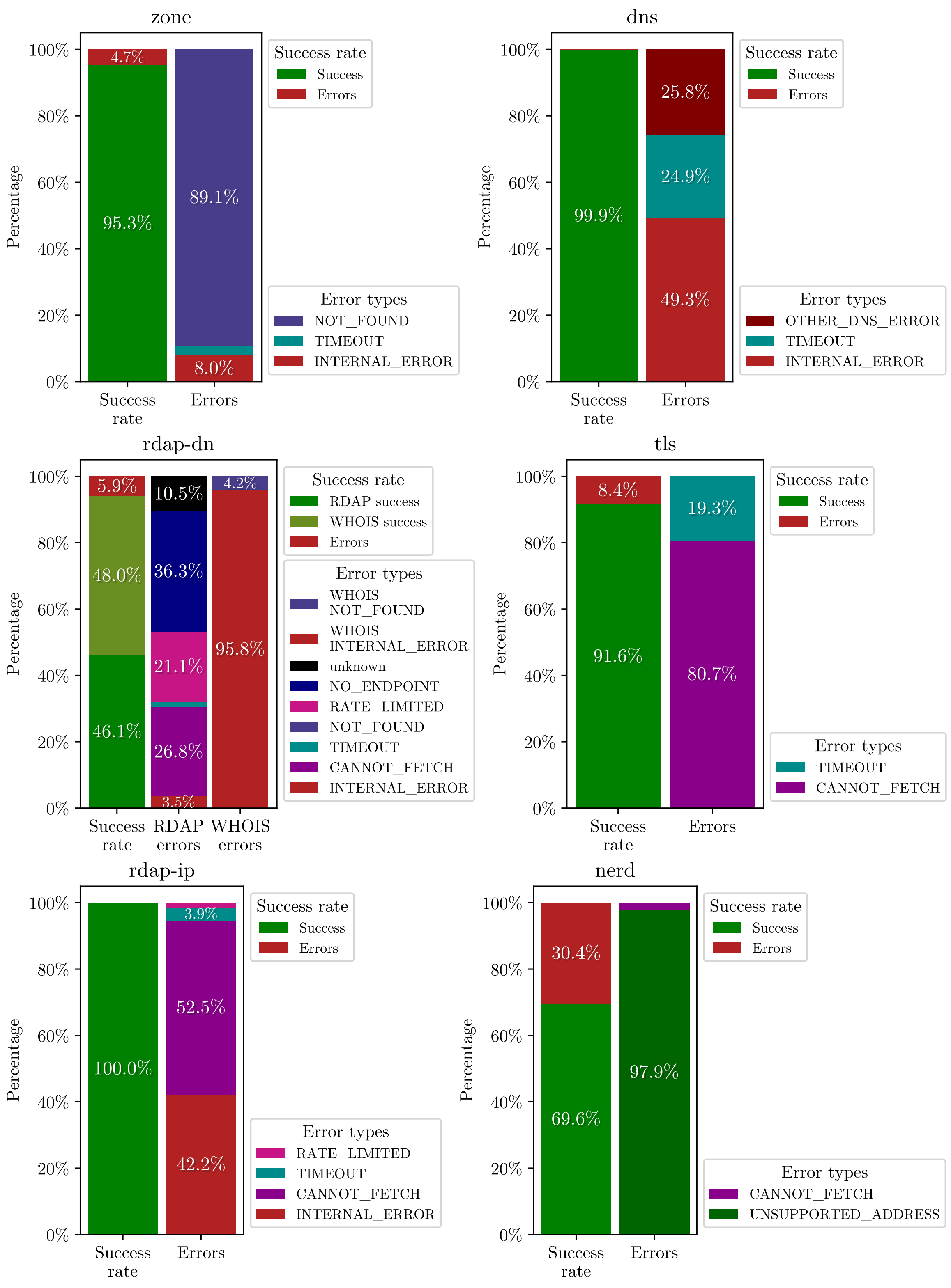


Figure H.2: Error types in experiment #1.

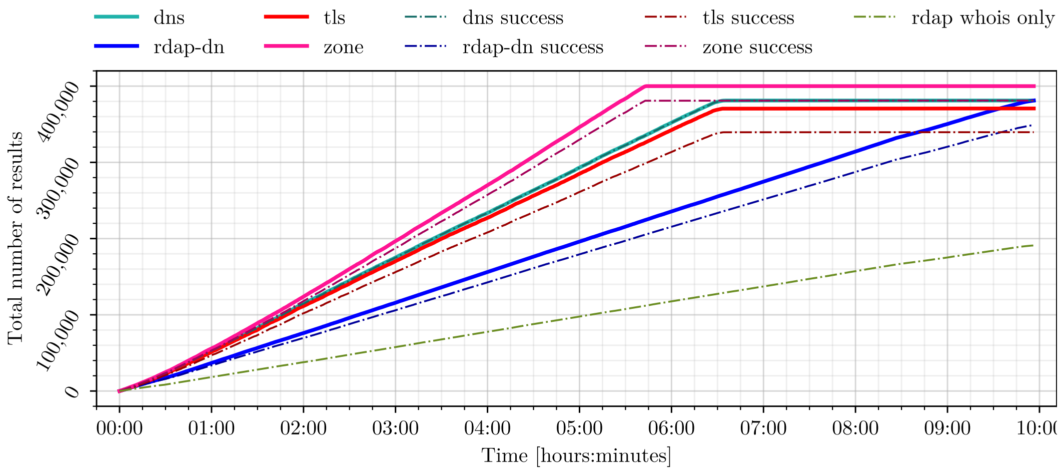
Experiment #2 (4 partitions)

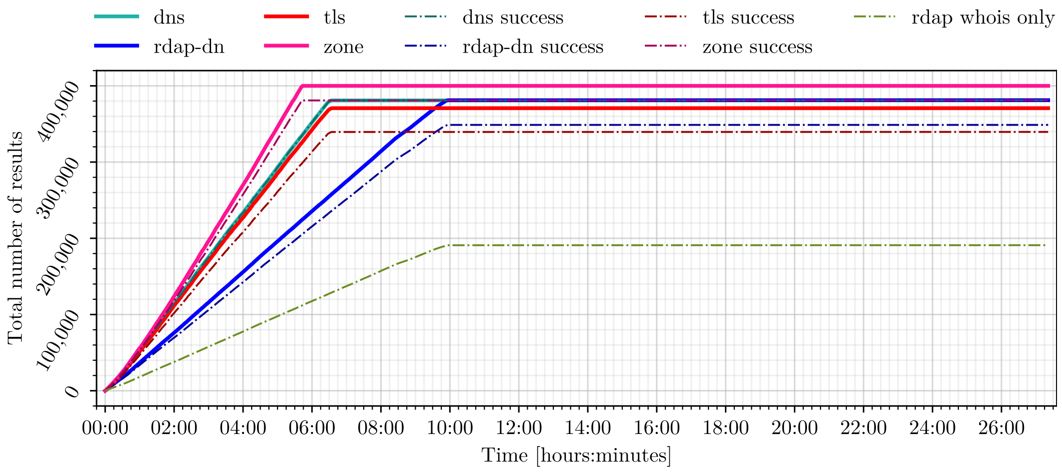
Collector	Tot		AvgTput		AvgColT		MnQdColT	
	[h:m]	Δ	[req/s]	Δ	[ms/req]	Δ	[s/req]	Δ
Zone	5:43	-33%	19.42	+50%	51	-34%	10 823	-42%
DNS	6:34	-53%	16.09	+114%	62	-53%	982	-88%
TLS	6:34	-53%	15.65	+114%	64	-53%	2.017	+445%
RDAP-DN	9:56	-50%	10.65	+99%	94	-50%	6 683	-69%
GEO-ASN	6:34	-53%	38.79	+114%	26	-53%	0.012	+50%
NERD	6:34	-53%	38.79	+114%	26	-53%	0.023	+28%
RDAP-IP	8:23	-56%	30.42	+126%	33	-56%	3 235	-68%
RTT	27:21*	-	6.18	-	162	-	69 340	-

Table H.3: Per-collector metrics in experiment #2, showing the differences to experiment #1. The RTT collector was again extremely slow in this test and was first reconfigured and then terminated prematurely, so the differences would be meaningless.

		Zone	DNS	TLS	R-DN	*WHOIS*	GEO	NERD	R-IP
NotFound	req%	4.21	0	0	0.01	0.42	0.04	29.74	0.00
	all%	4.21	0	0	0.01	-	-	-	-
Remote	req%	0.13	0.03	8.40	16.93	0	0	0	0.03
	all%	0.13	0.02	7.79	16.13	-	-	-	-
Internal	req%	0.37	0.01	0	1.89	14.08	0	0	0.02
	all%	0.37	0.01	0	1.80	-	-	-	-
RateLimit	req%	-	-	-	13.76	0	-	-	0.06
	all%	-	-	-	13.11	-	-	-	-
NoEndpt	req%	-	-	-	17.75	0	-	-	0
	all%	-	-	-	16.92	-	-	-	-
Other	req%	0	0	0	8.25	0	0	0	0
	all%	0	0	0	7.86	-	-	-	-
Total	req%	4.72	0.03	8.40	58.59	14.49	0.04	29.74	0.11
	all%	4.72	0.03	7.79	55.83	0	0	0	0

Table H.4: Error rates in experiment #2.





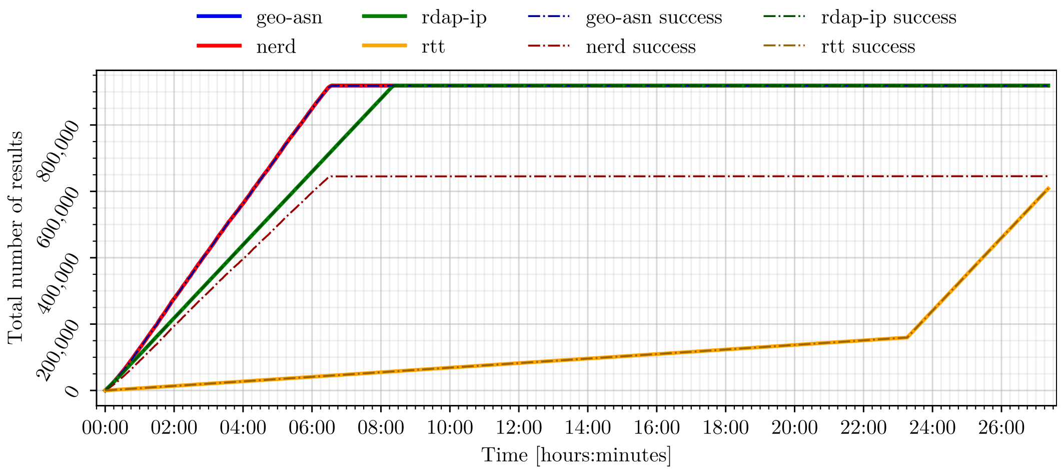


Figure H.3: Number of responses over time in experiment #2. The chart for the DN-based collectors is shown twice: the top one is scoped to the runtime of the DN-based collectors only, the bottom one shows the entire test duration to provide better comparison with the IP-based collectors in the bottom chart.

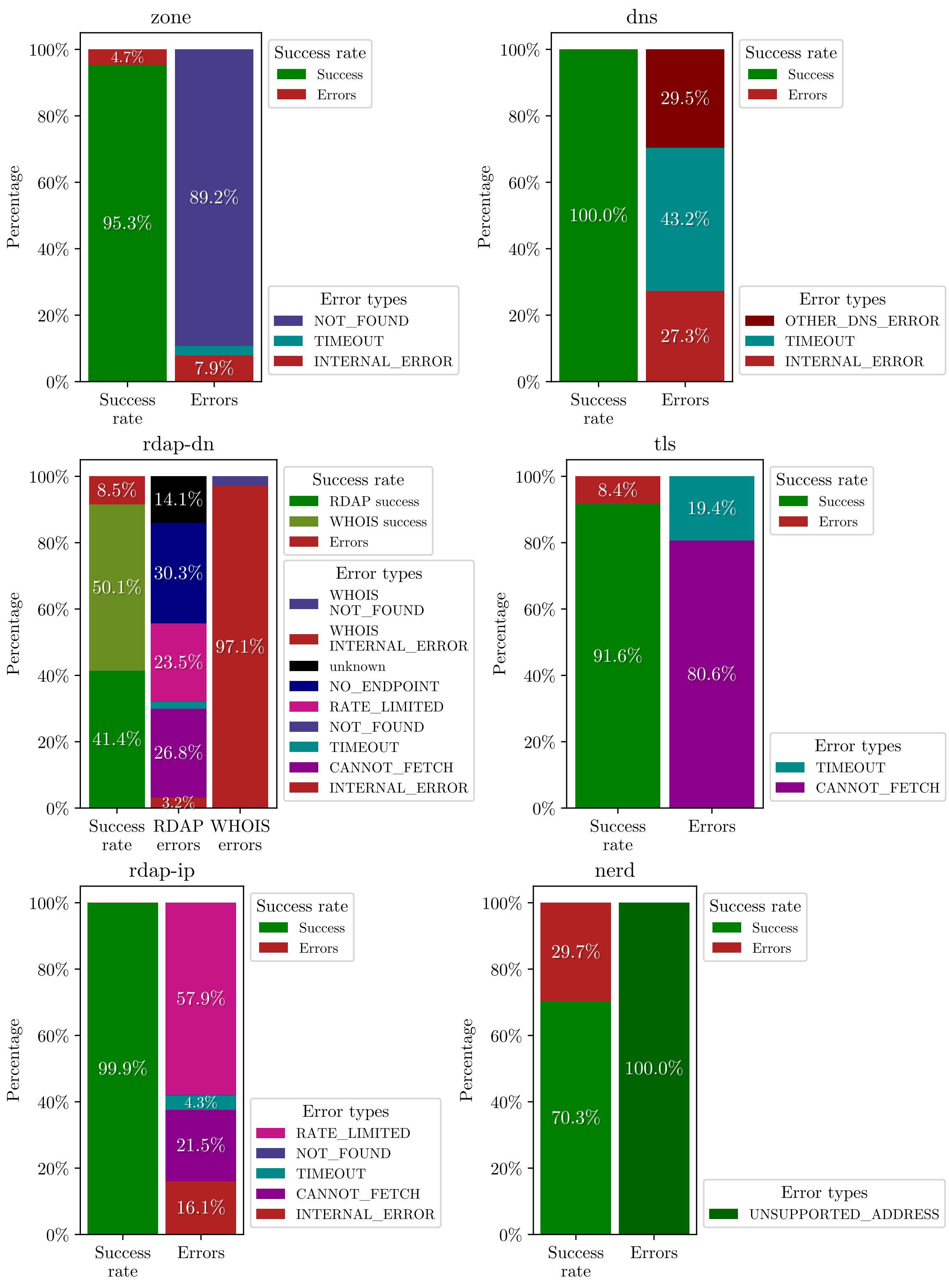


Figure H.4: Error types in experiment #2.

Experiment #3 (8 partitions)

Collector	Tot		AvgTput		AvgColT		MnQdColT	
	[h:m]	Δ	[req/s]	Δ	[ms/req]	Δ	[s/req]	Δ
Zone	5:29	-4%	20.23	+4%	49	-4%	10 443	-4%
DNS	5:29	-16%	19.24	+20%	52	-16%	2.194	+123%
TLS	5:29	-16%	18.72	+20%	53	-17%	0.279	-86%
RDAP-DN	5:58	-40%	17.72	+66%	56	-40%	486	-93%
GEO-ASN	5:29	-16%	46.43	+20%	22	-16%	0.023	+92%
NERD	5:29	-16%	46.43	+20%	22	-16%	0.033	+43%
RDAP-IP	5:29	-35%	46.43	+53%	22	-33%	0.933	-71%
RTT	5:29	-	46.42	-	22	-	8.603	-

Table H.5: Per-collector metrics in experiment #3, showing the differences to experiment #2.

		Zone	DNS	TLS	R-DN	*WHOIS*	GEO	NERD	R-IP
NotFound	req%	4.22	0	0	0.07	0.38	0.03	29.75	0.00
	all%	4.22	0	0	0.06	-	-	-	-
Remote	req%	0.15	0.02	8.65	17.94	0	0	0	0.02
	all%	0.15	0.02	8.01	17.09	-	-	-	-
Internal	req%	0.40	0.01	0	1.85	17.69	0	0	0.06
	all%	0.40	0.01	0	1.76	-	-	-	-
RateLimit	req%	-	-	-	12.87	0	-	-	0.01
	all%	-	-	-	12.26	-	-	-	-
NoEndpt	req%	-	-	-	16.84	0	-	-	0
	all%	-	-	-	16.04	-	-	-	-
Other	req%	0	0	0	10.66	0	0	0	0
	all%	0	0	0	10.15	-	-	-	-
Total	req%	4.77	0.03	8.65	60.23	18.07	0.03	29.75	0.09
	all%	4.77	0.03	8.01	57.36	0	0	0	0

Table H.6: Error rates in experiment #3.

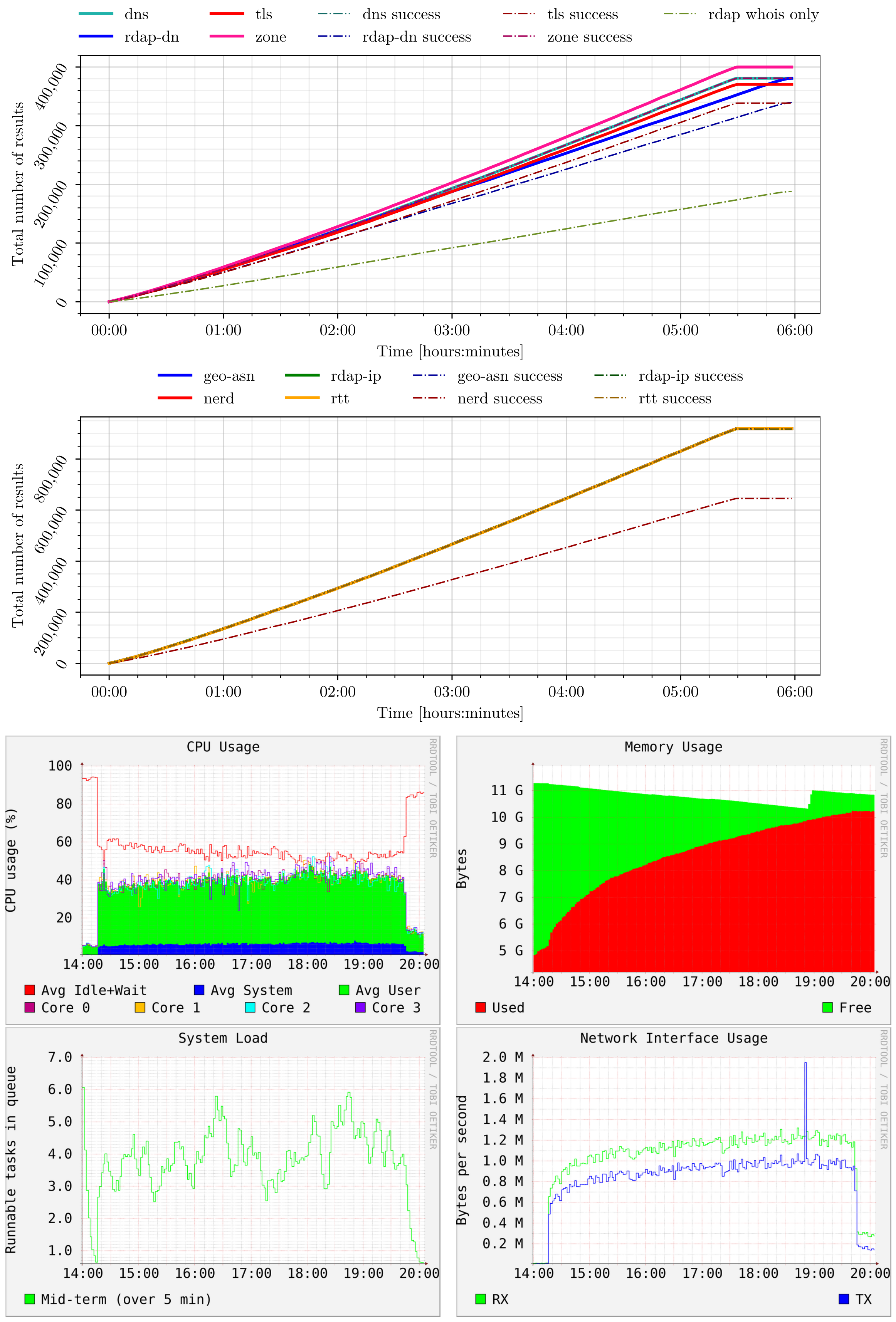


Figure H.5: Number of responses and resource usage over time in experiment #3.

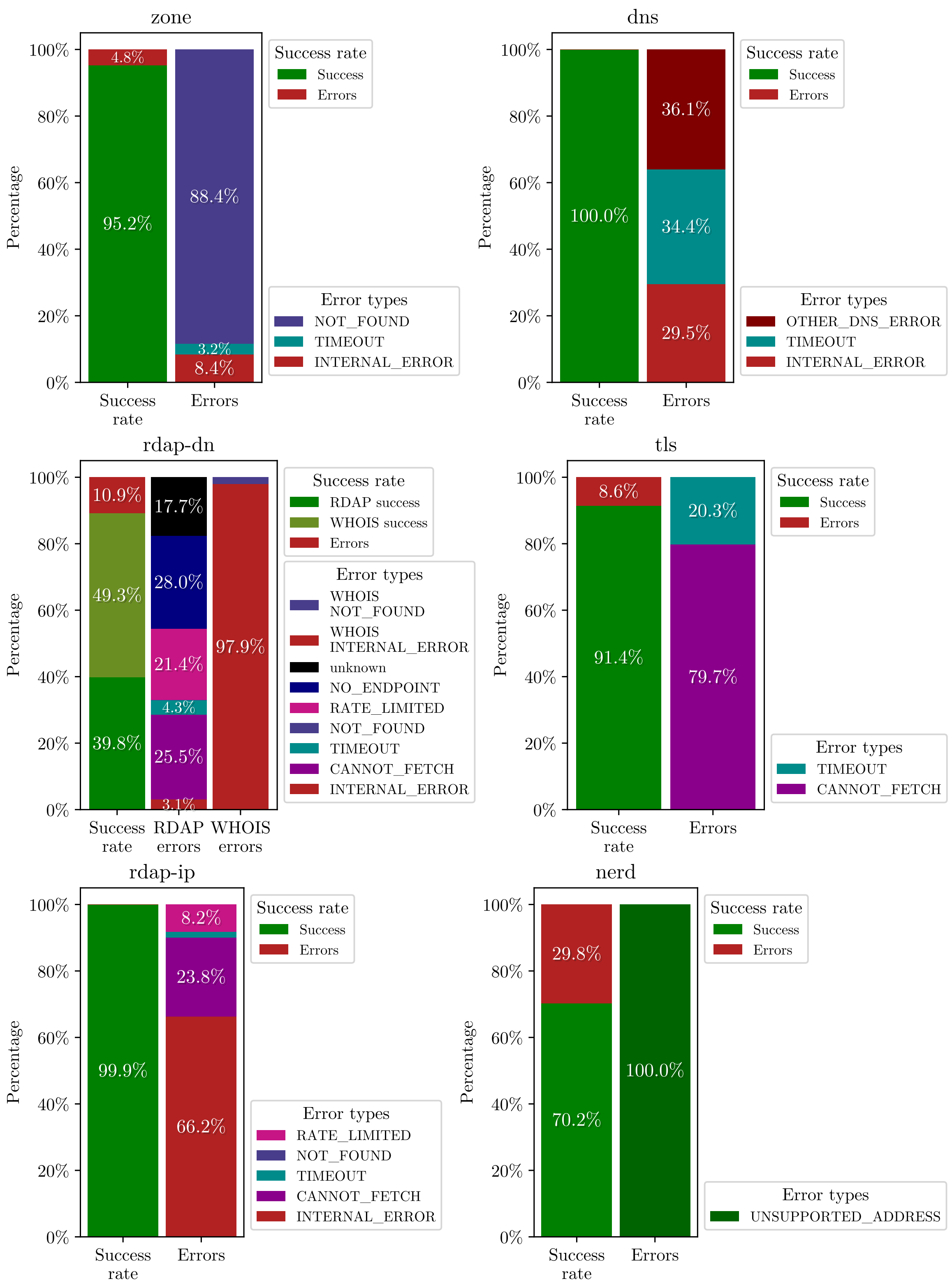


Figure H.6: Error types in experiment #3.

Experiment #4 (12 partitions)

Collector	Tot		AvgTput		AvgColT		MnQdColT	
	[h:m]	Δ	[req/s]	Δ	[ms/req]	Δ	[s/req]	Δ
Zone	3:04	-44%	36.29	+79%	28	-43%	5 133	-51%
DNS	3:04	-44%	31.44	+63%	32	-38%	0.960	-56%
TLS	3:04	-44%	30.59	+63%	33	-38%	0.337	+21%
RDAP-DN	4:16	-28%	22.65	+28%	44	-21%	2 489	+412%
GEO-ASN	3:04	-44%	75.78	+63%	13	-41%	0.044	+91%
NERD	3:04	-44%	75.78	+63%	13	-41%	0.141	+327%
RDAP-IP	3:04	-44%	75.76	+63%	13	-41%	1.837	+97%
RTT	3:04	-44%	75.75	+63%	13	-41%	8.425	-2%

Table H.7: Per-collector metrics in experiment #4.

		Zone	DNS	TLS	R-DN	*WHOIS*	GEO	NERD	R-IP
NotFound	req%	4.02	0	0	0.01	0.33	0.03	29.84	0
	all%	4.03	0	0	0.01	-	-	-	-
Remote	req%	9.13	0.02	8.49	18.62	0	0	0.01	0.04
	all%	9.16	0.02	7.19	16.21	-	-	-	-
Internal	req%	0.15	0.01	0	1.84	17.56	0	0.01	0.06
	all%	0.15	0.01	0	1.60	-	-	-	-
RateLimit	req%	-	-	-	14.56	0	-	-	0.20
	all%	-	-	-	12.67	-	-	-	-
NoEndpt	req%	-	-	-	16.65	0	-	-	0
	all%	-	-	-	14.49	-	-	-	-
Other	req%	0	0	0	11.01	0	0	0	0
	all%	0	0	0	9.58	-	-	-	-
Total	req%	13.29	0.03	8.49	62.68	17.89	0.03	29.85	0.29
	all%	13.35	0.03	7.19	54.56	0	0	0	0

Table H.8: Error rates in experiment #4.

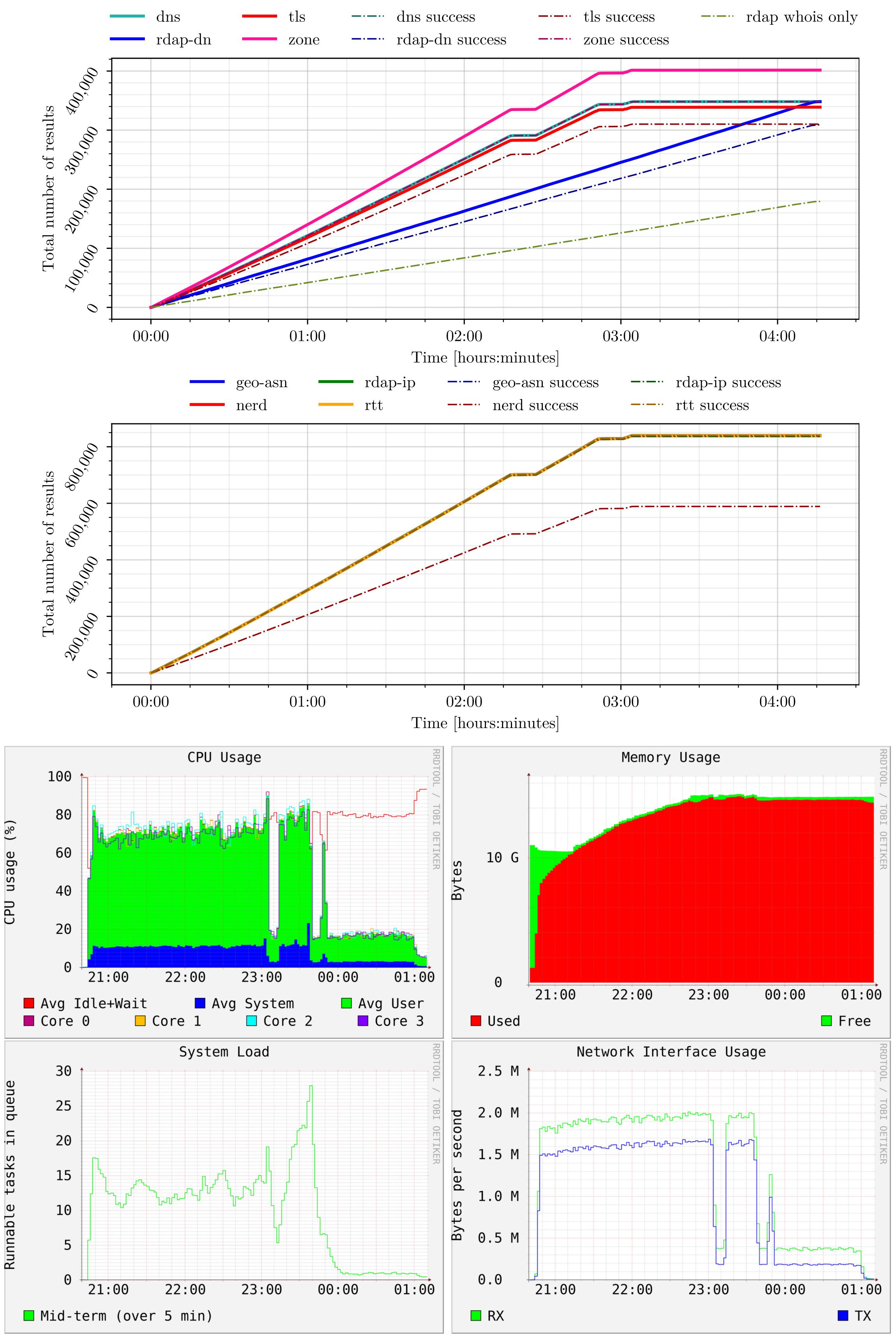


Figure H.7: Number of responses and resource usage over time in experiment #4.

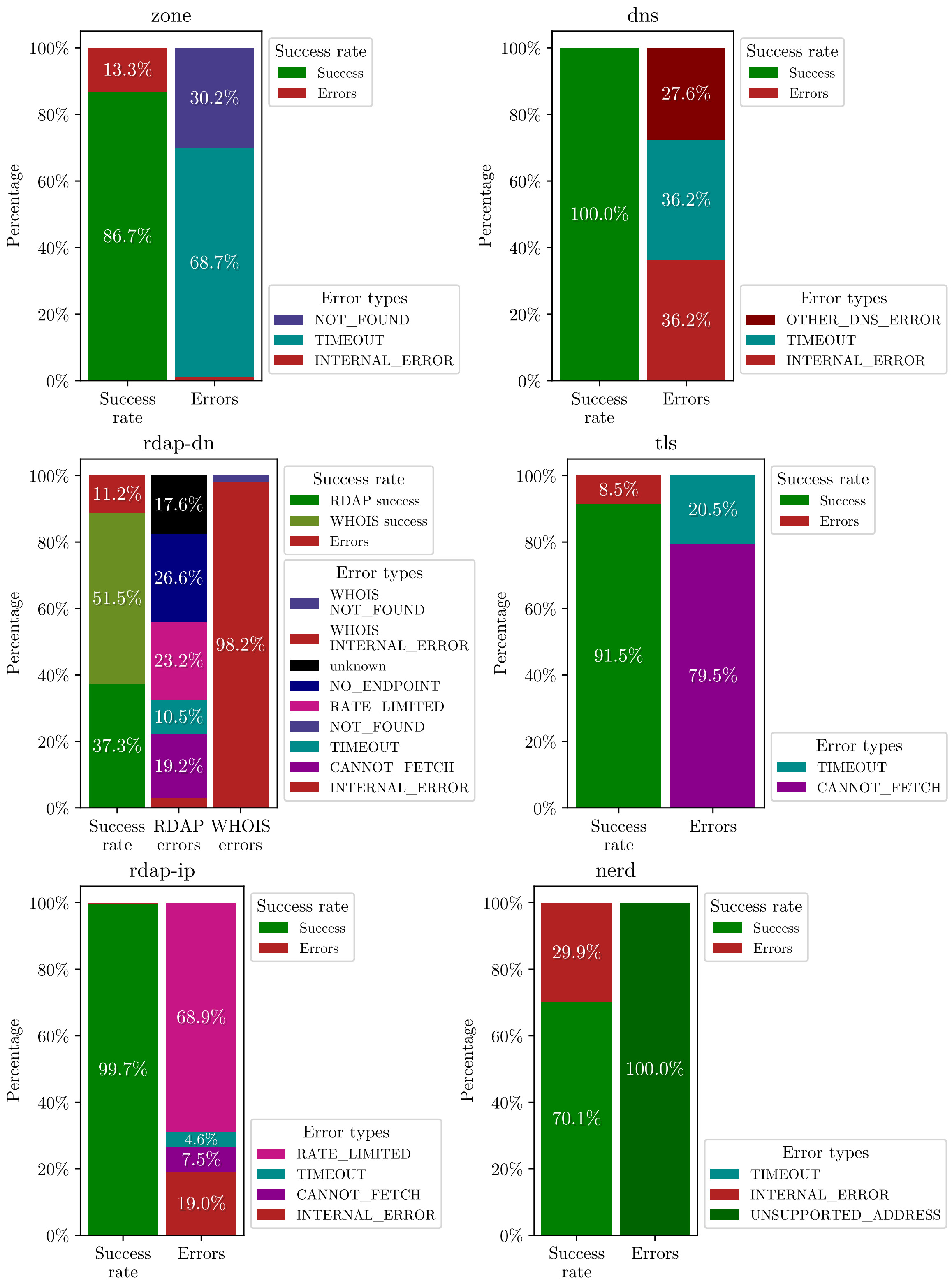


Figure H.8: Error types in experiment #4.

Load on the Broker

	infra		*scanner*	
Tot [min]	$\overline{\text{CPU}}$ [%]	$\overline{\text{Mem}}$ [GiB]	$\overline{\text{CPU}}$ [%]	$\overline{\text{Mem}}$ [GiB]
200	17	3.83	62	11.39

Metrics of the broker load experiment.

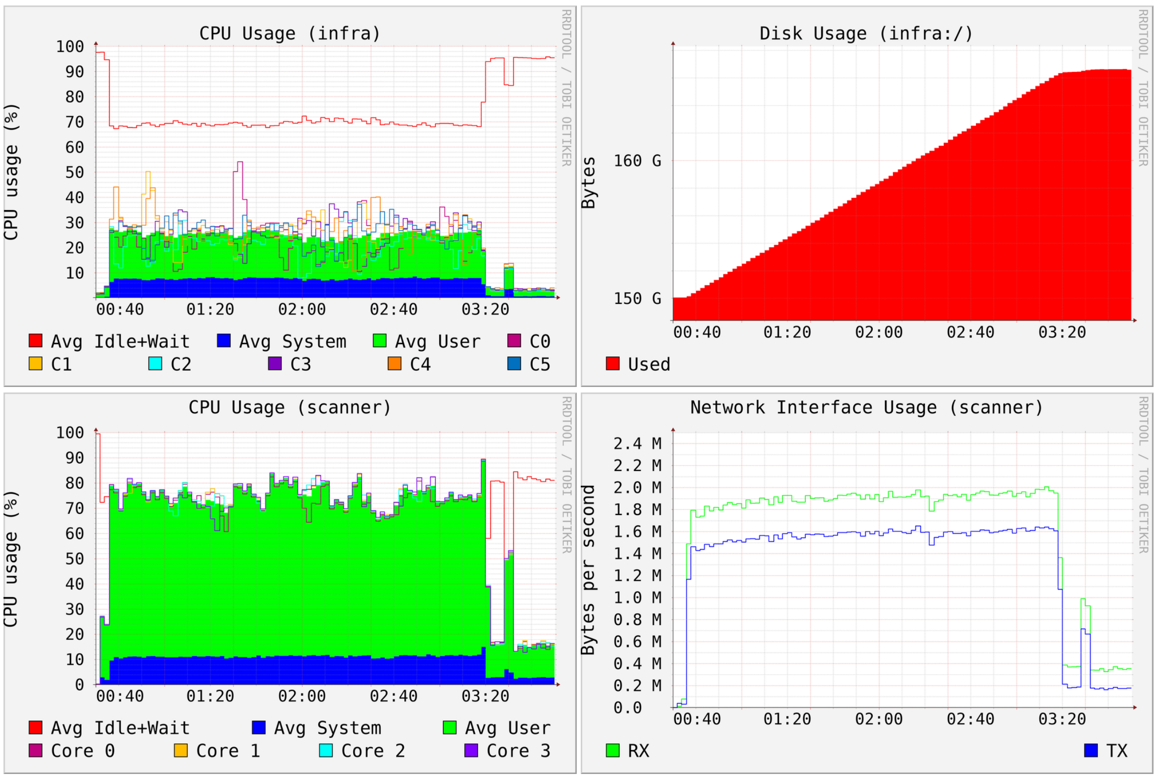


Figure H.9: Comparison of the load on both VMs during collection.

H.2 Data Merger Experiments

Test	Tot [min]	MnProcT [ms]	MTput [DN/s]	$\overline{\text{CPU}}$ [%]	$\overline{\text{Mem}}$ [GiB]
#1	33	5.69	175.61	**47**	36.63
#2	25	4.31	231.81	59	21.03
#3	**24**	**4.14**	**241.47**	57	**18.08**

Metrics of the three merger runs. The best values are in bold.

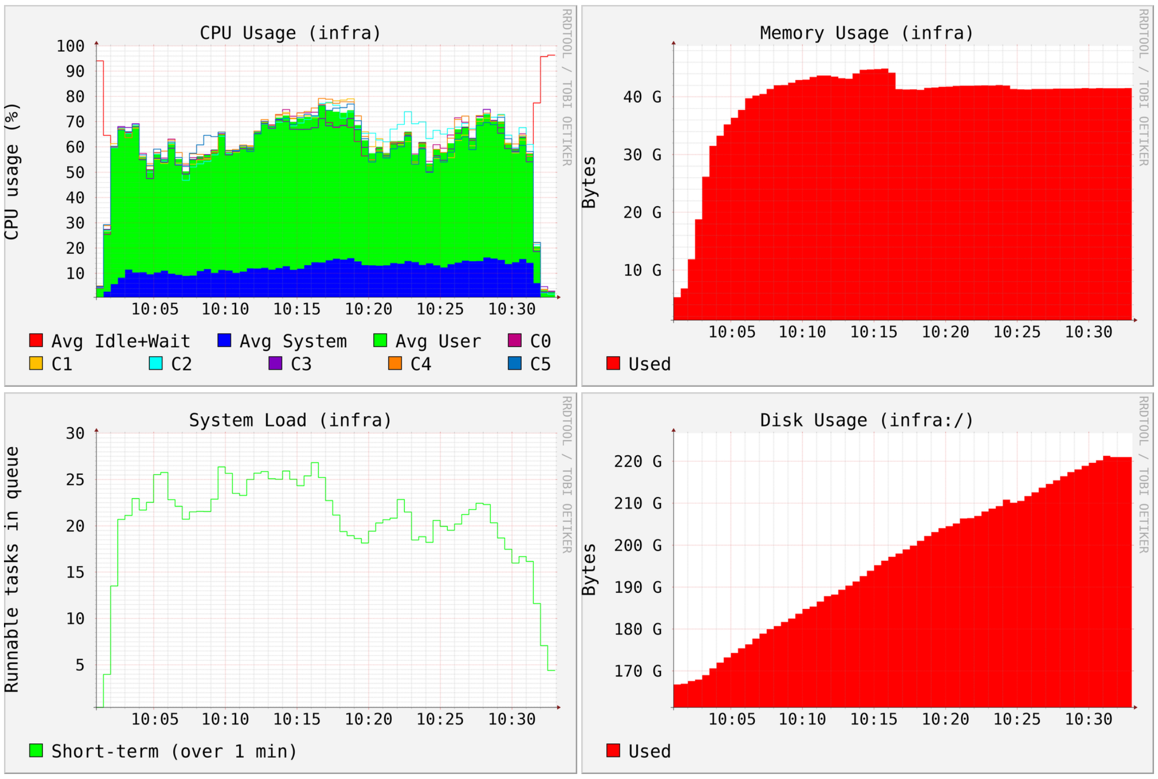


Figure H.10: Data merger system resources usage in the test with $M = 4, M^t = 4$.

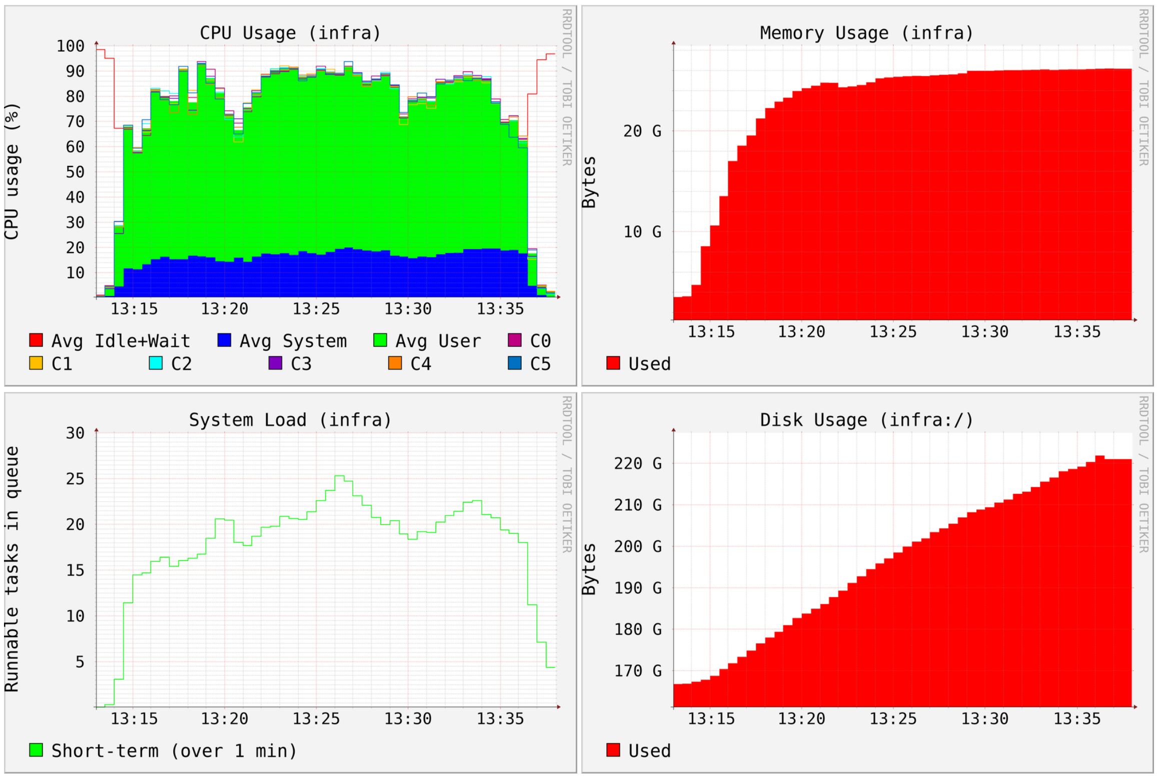


Figure H.11: Data merger system resources usage in the test with $M = 1, M^t = 16$.

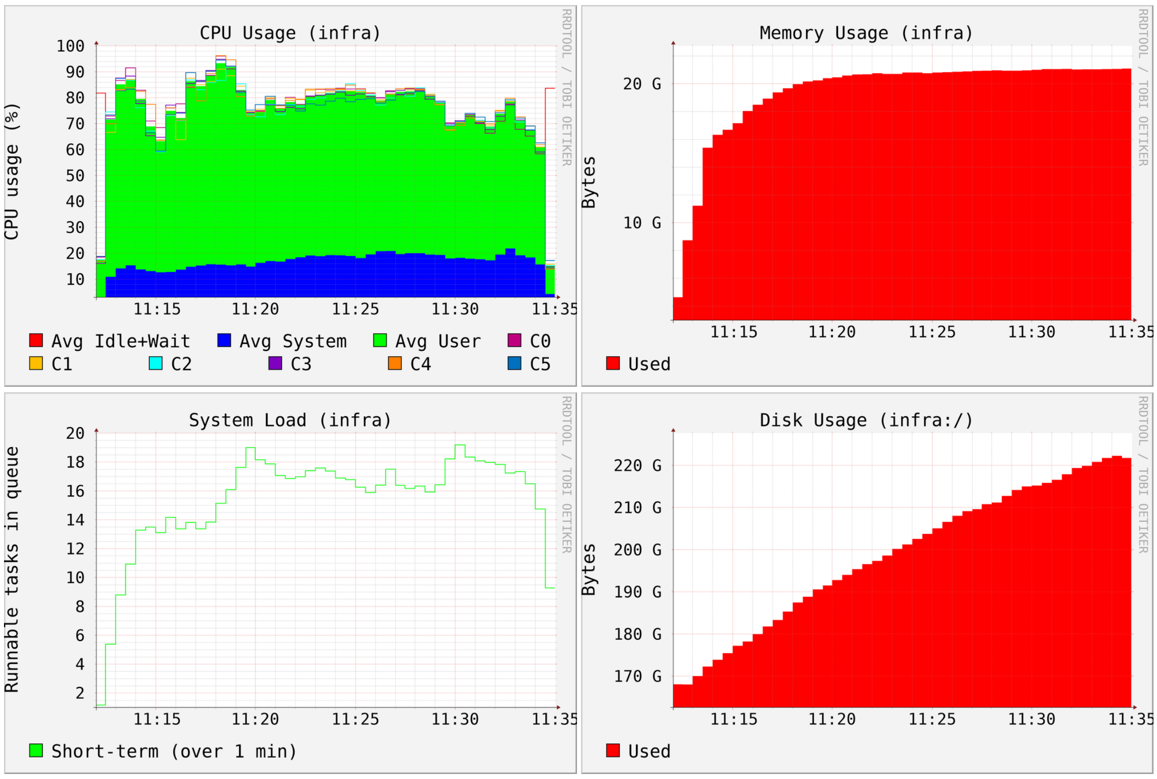


Figure H.12: Data merger system resources usage in the test with $M = 1, M^t = 8$.

H.3 Feature Extractor Experiments

	Tot [min]	$\overline{\text{CPU}}$ [%]	$\overline{\text{Mem}}$ [GiB]	Tot [min]	$\overline{\text{CPU}}$ [%]	$\overline{\text{Mem}}$ [GiB]	Tot [min]	$\overline{\text{CPU}}$ [%]	$\overline{\text{Mem}}$ [GiB]
Inst. E	$E^b = 200,\ E^t = 4$			$E^b = 200,\ E^t = 8$			$E^b = 200,\ E^t = 16$		
1	22	46	**3.276**	22	**45**	3.954	19	46	5.162
4	10	78	6.520	11	78	8.671		—	
Inst. E	$E^b = 50$			$E^b = 100$			$E^b = 200$		
8	11	71	4.025	**9**	76	4.037	**9**	66	4.097
16	15	67	5.244	12	65	5.151	**9**	72	5.354

System metrics during the feature extractor experiments.

1 partition, multiprocessing

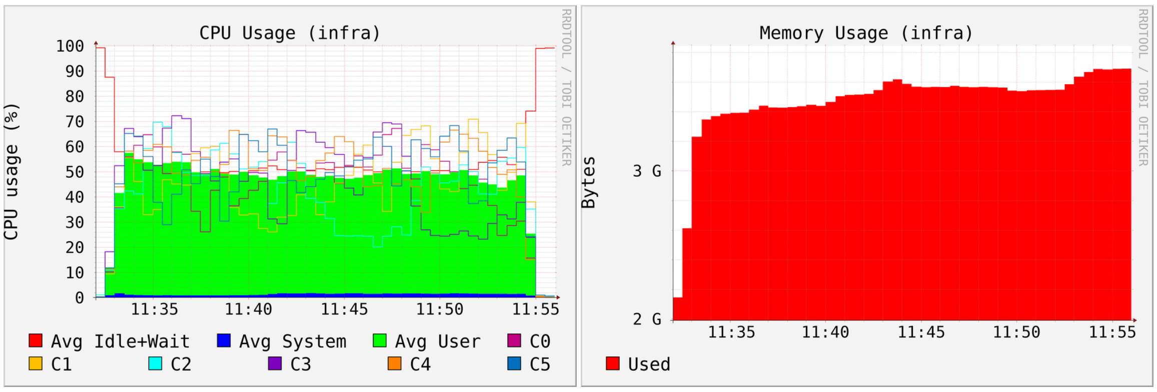


Figure H.13: System resources usage in the test with $E = 1, E^b = 200, E^t = 4$.

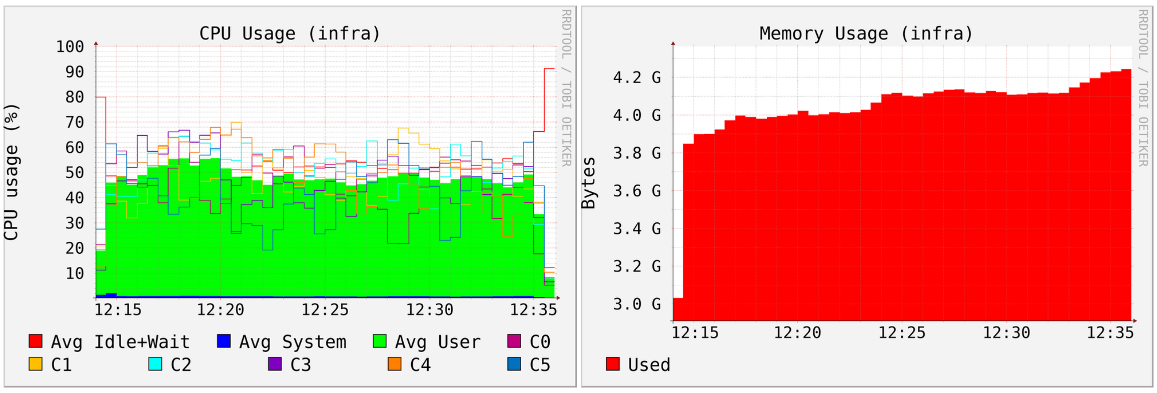


Figure H.14: System resources usage in the test with $E = 1, E^b = 200, E^t = 8$.

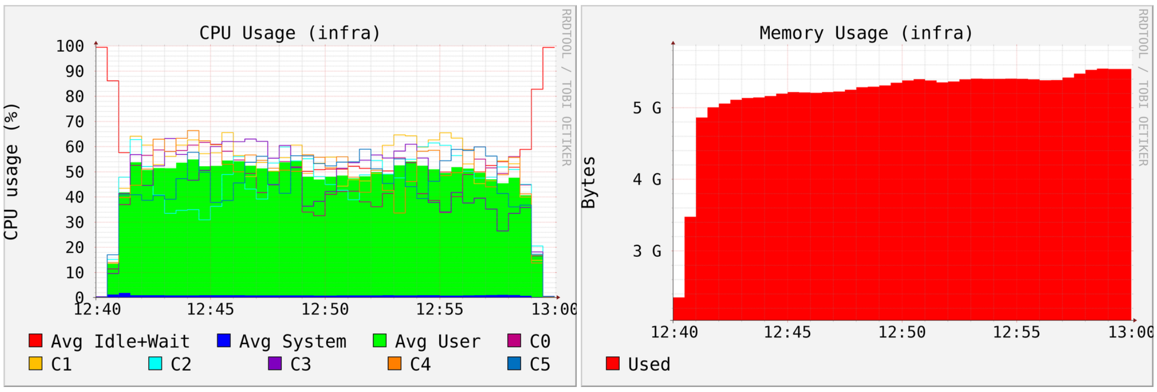


Figure H.15: System resources usage in the test with $E = 1, E^b = 200, E^t = 16$.

4 partitions, multiprocessing

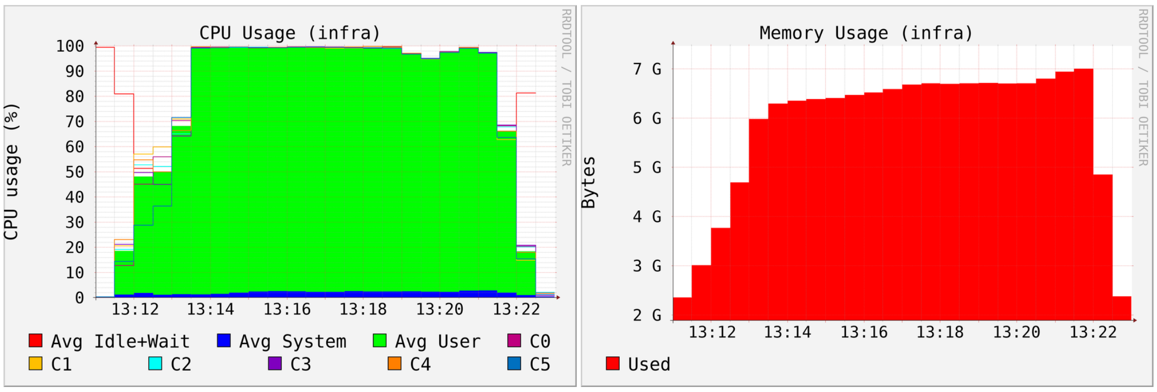
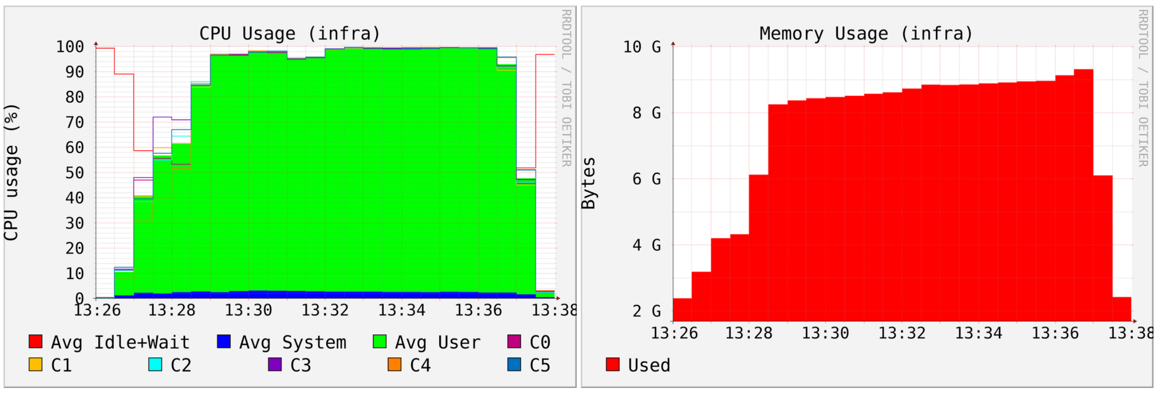


Figure H.16: System resources usage in the test with $E = 4, E^b = 200, E^t = 4$.



System resources usage in the test with $E = 4, E^b = 200, E^t = 8$.

8 partitions

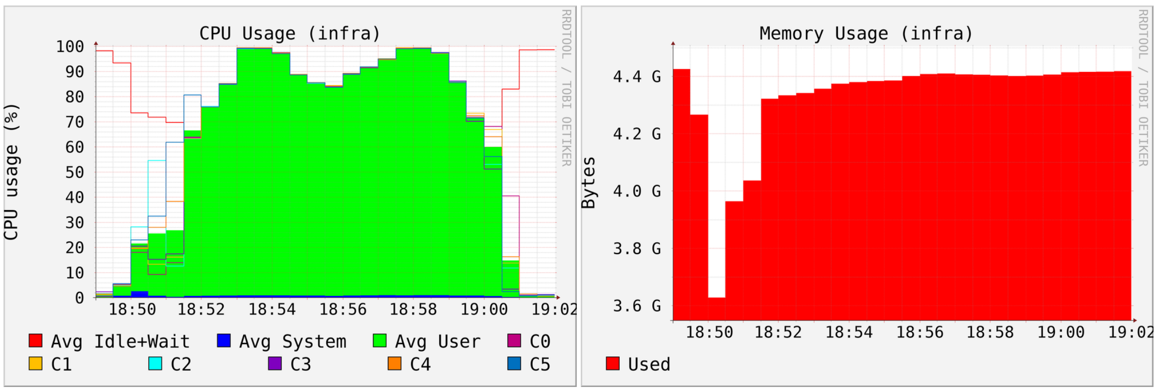


Figure H.17: Extractor system resources usage in the test with $E = 8, E^b = 50, E^t = 1$.

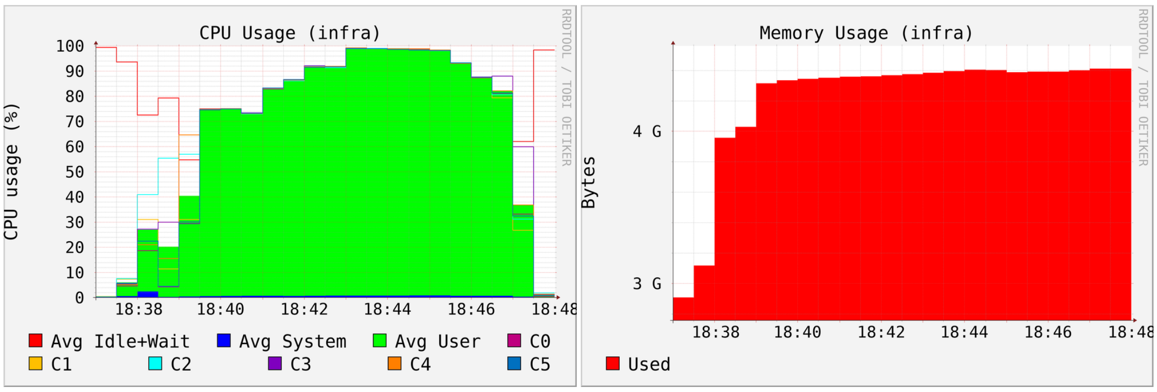


Figure H.18: Extractor system resources usage in the test with $E = 8, E^b = 100, E^t = 1$.

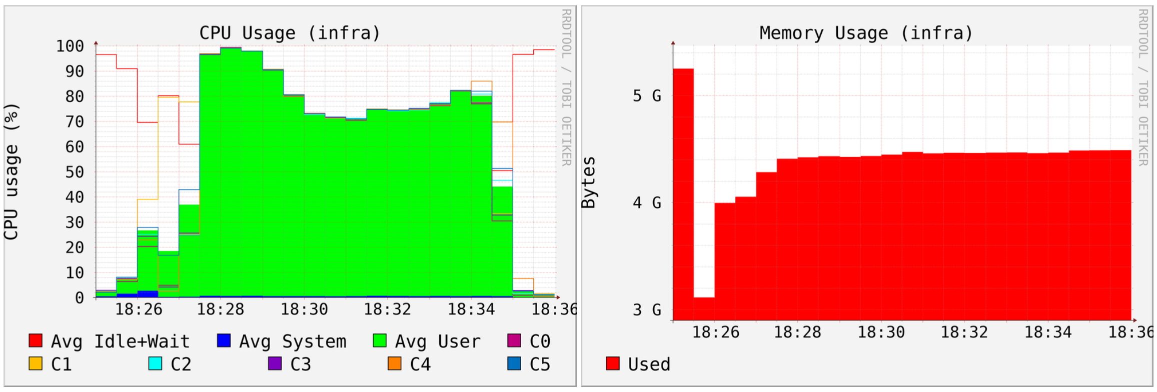


Figure H.19: Extractor system resources usage in the test with $E = 8, E^b = 200, E^t = 1$.

16 partitions

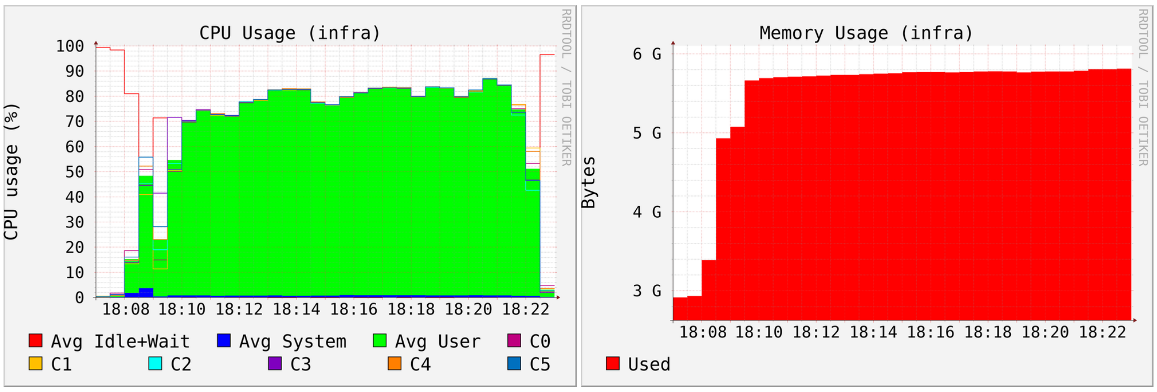


Figure H.20: System resources usage in the test with $E = 16, E^b = 50, E^t = 1$.

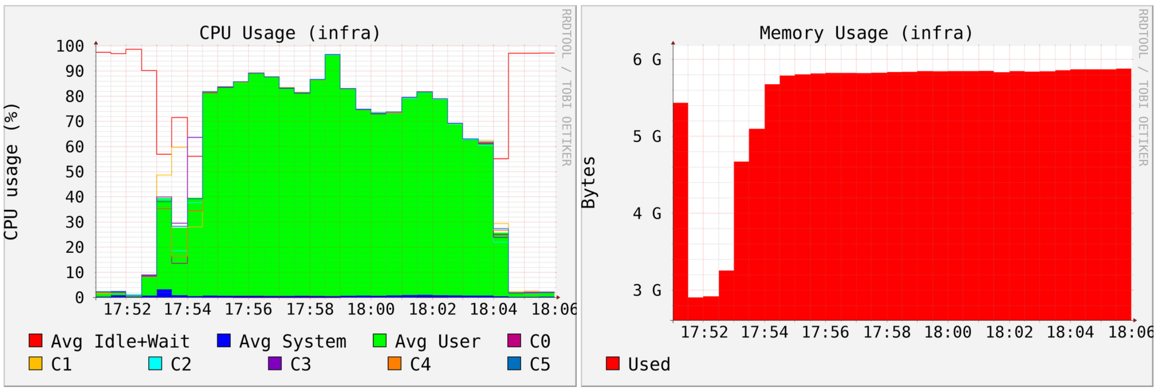


Figure H.21: System resources usage in the test with $E = 16, E^b = 100, E^t = 1$.

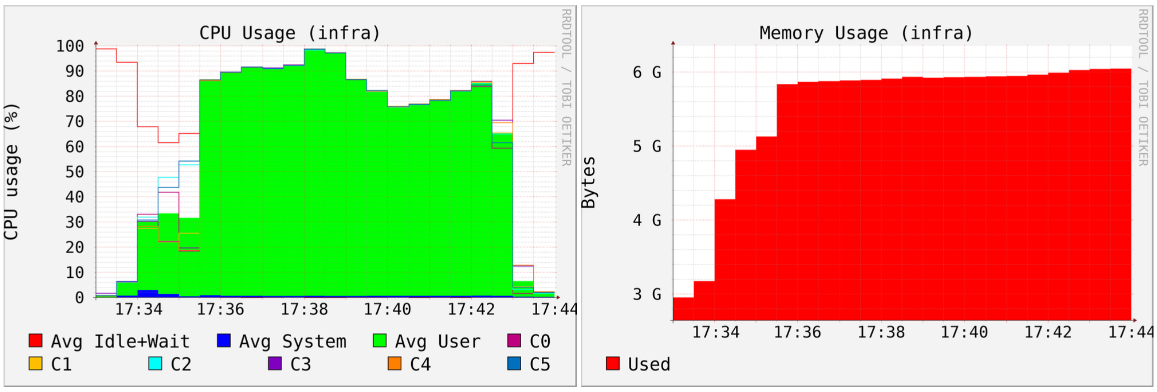


Figure H.22: System resources usage in the test with $E = 16, E^b = 200, E^t = 1$.

H.4 Real-time Processing Experiments

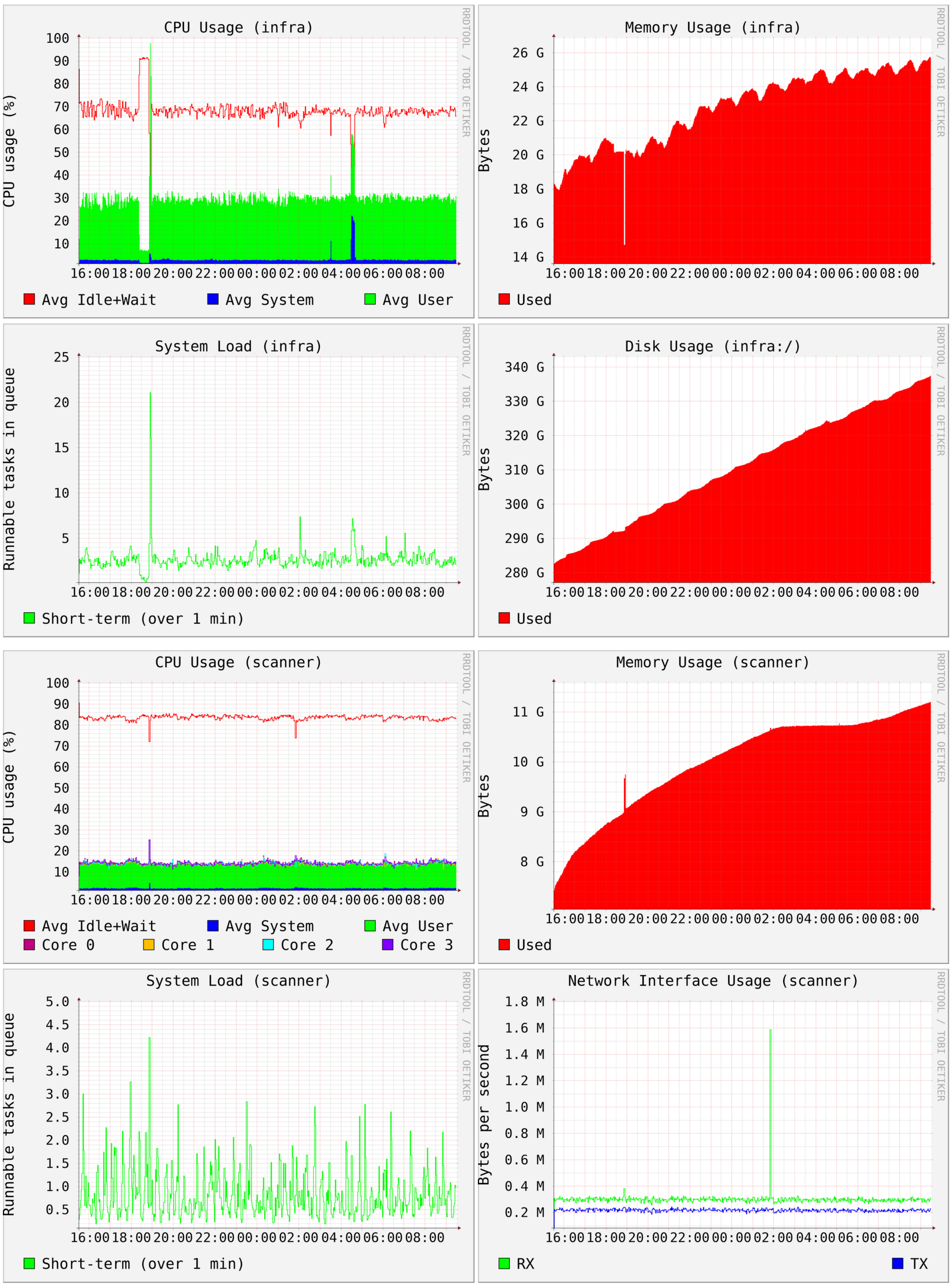


Figure H.23: System resources usage of the two VMs in experiment #1.

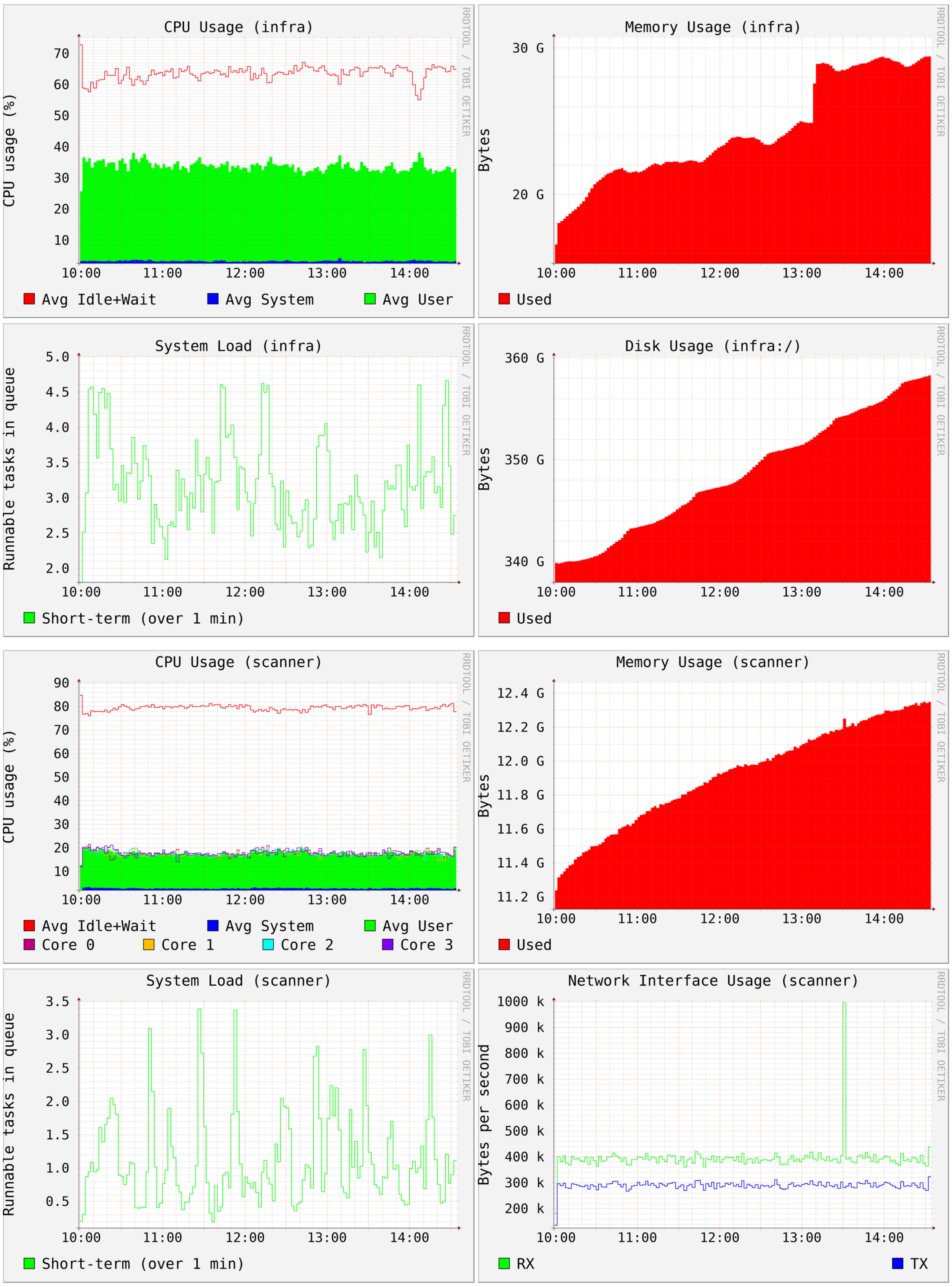


Figure H.24: System resources usage of the two VMs in experiment #2.

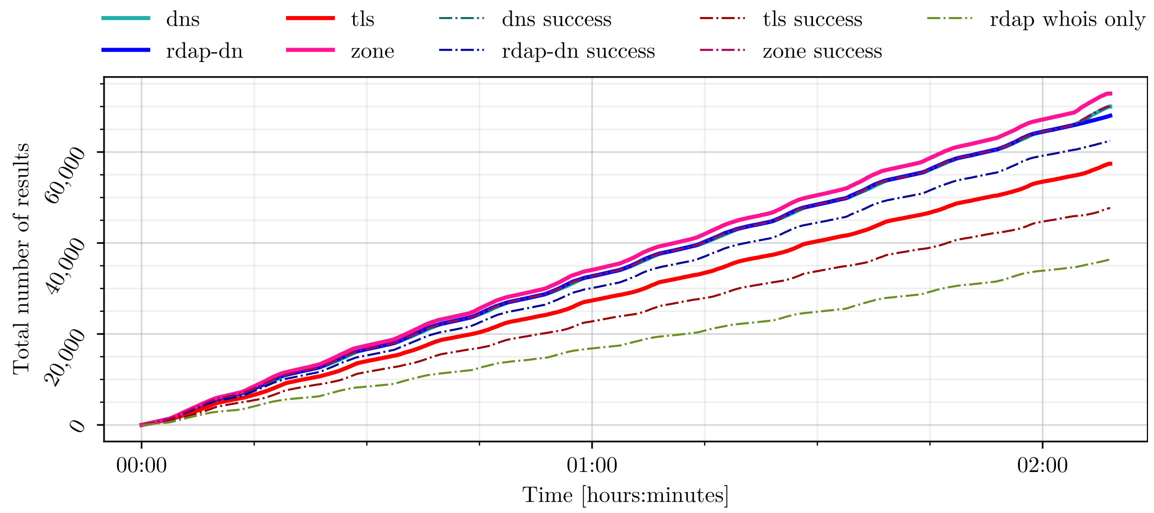
Real-time Experiment #3

Collector	Tot [h:m]	AvgTput [req/s]	AvgColT [ms/req]	MnQdColT [s/req]
Zone	02:08	9.43	106	-
DNS	02:08	9.04	110	11.23
RDAP-DN	02:08	8.79	113	4.96
TLS	02:08	7.42	134	0.46
GEO-ASN	02:08	18.19	54	0.01
NERD	02:08	18.19	54	0.02
RDAP-IP	02:08	18.20	54	0.52
RTT	02:08	18.20	54	6.54

Table H.9: Per-collector metrics in real-time experiment #3.

		Zone	DNS	TLS	R-DN	*WHOIS*	GEO	NERD	R-IP
NotFound	req%	1.92	0	0	0.01	0.58	0.21	30.55	0.00
Remote	req%	1.15	0.02	16.88	11.46	0	0	0.01	0.41
Internal	req%	0.86	0	0.13	0.08	12.55	0	0	0.92
RateLimit	req%	-	-	-	13.54	0	-	-	0.18
NoEndpt	req%	-	-	-	28.77	0	-	-	0
Other	req%	0	0	0	7.73	0	0	0	0
Total	req%	3.93	0.02	17.01	61.59	13.12	0.21	30.56	1.51

Table H.10: Error rates in real-time experiment #3.



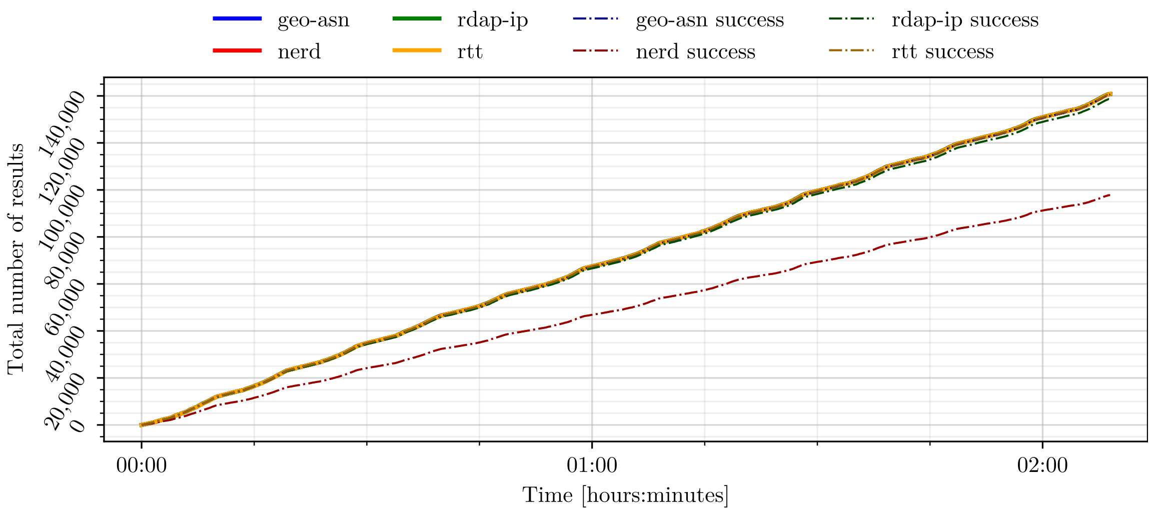


Figure H.25: Number of responses over time in real-time experiment #3.

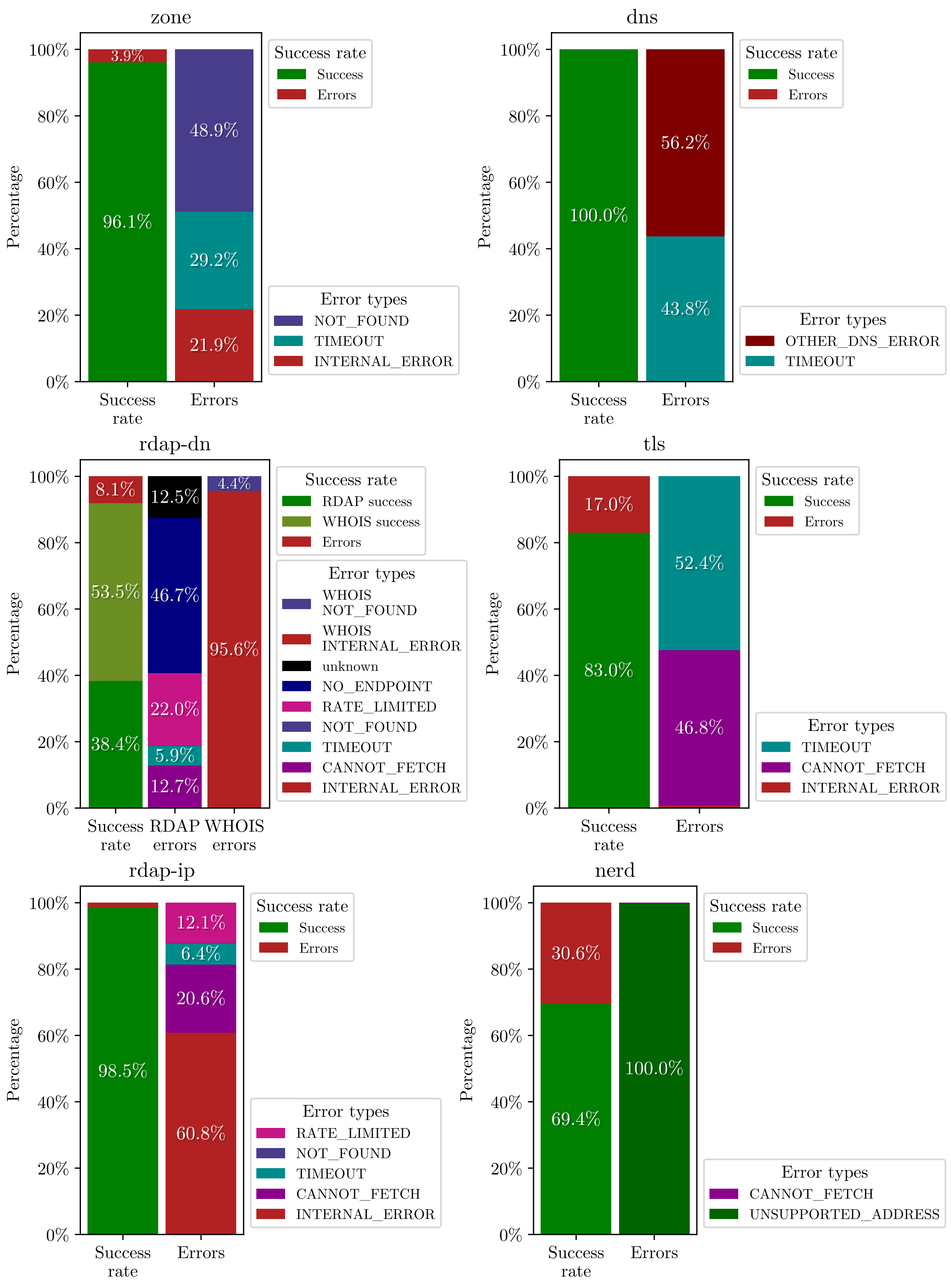


Figure H.26: Error types in real-time experiment #3.

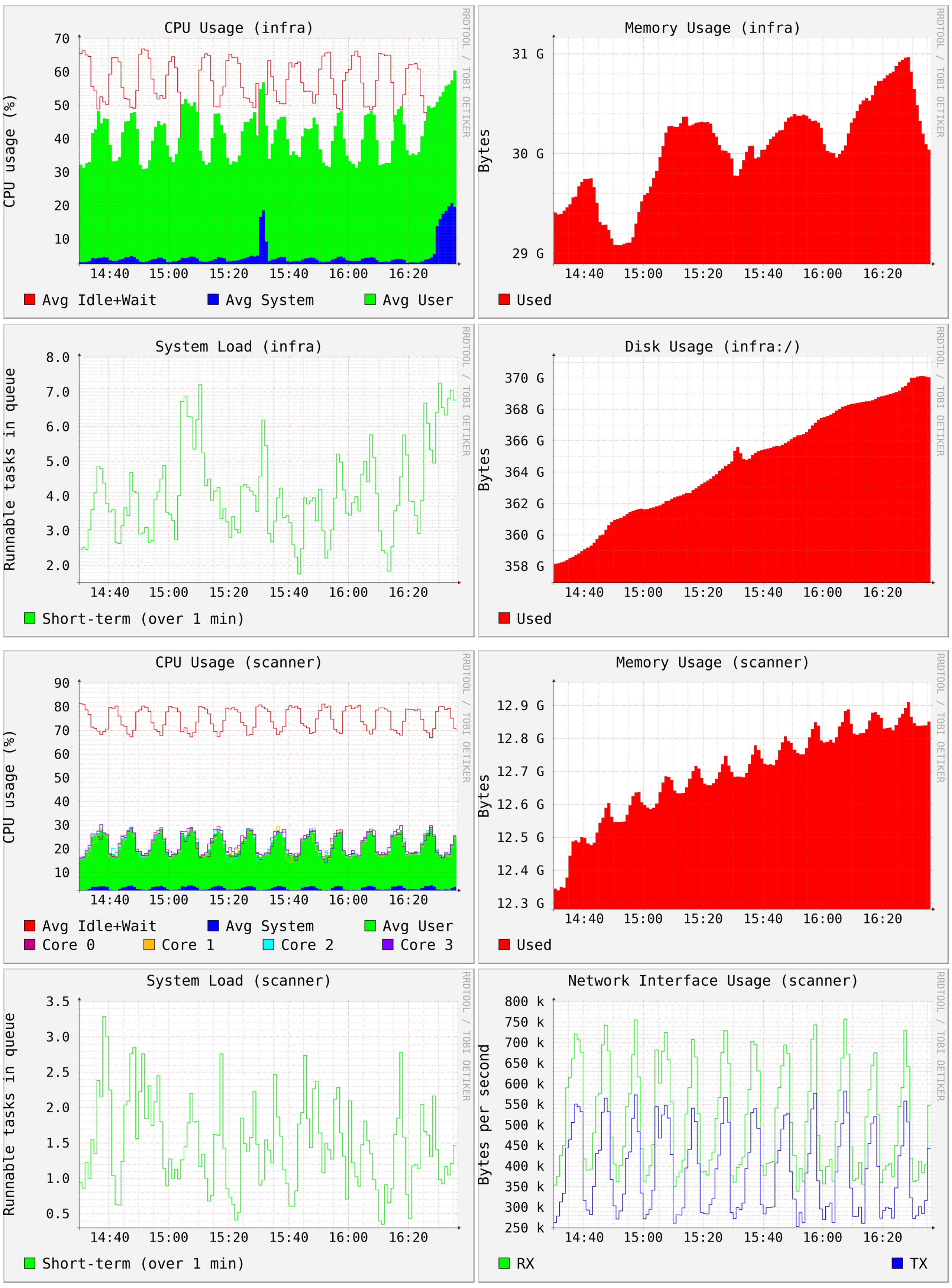


Figure H.27: System resources usage of the two VMs in experiment #3.

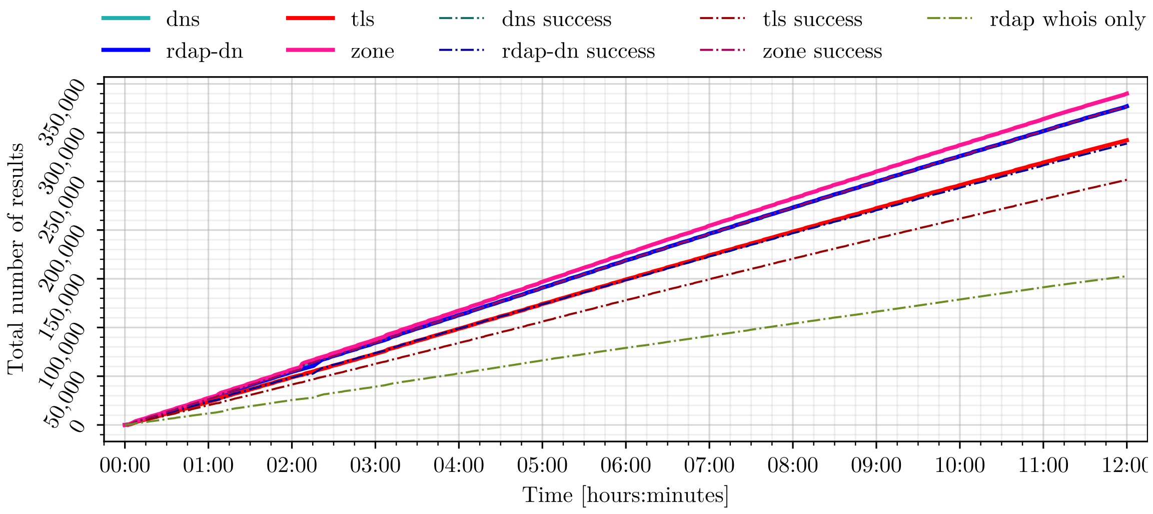
Real-time Experiment #4

Collector	Tot [h:m]	AvgTput [req/s]	AvgColT [ms/req]	MnQdColT [s/req]
Zone	11:59	7.87	127	-
DNS	11:59	7.57	132	4.06
TLS	11:59	6.76	147	0.53
RDAP-DN	11:59	7.57	132	12.63
GEO-ASN	11:59	15.76	63	0.01
NERD	11:59	15.76	63	0.02
RDAP-IP	11:59	15.76	63	0.42
RTT	11:59	15.76	63	6.32

Table H.11: Per-collector metrics in real-time experiment #4.

		Zone	DNS	TLS	R-DN	*WHOIS*	GEO	NERD	R-IP
NotFound	req%	2.81	0	0	0.03	0.53	0.22	29.39	0.00
Remote	req%	1.03	0.01	13.85	12.95	0.00	0	0.00	0.50
Internal	req%	0	0	0.03	0.13	19.51	0	0	1.07
RateLimit	req%	-	-	-	21.58	0	-	-	0.35
NoEndpt	req%	-	-	-	23.69	0	-	-	0
Other	req%	0	0	0	0	0	0	0	0
Total	req%	3.84	0.01	13.88	58.38	20.04	0.22	29.39	1.92

Table H.12: Error rates in real-time experiment #4.



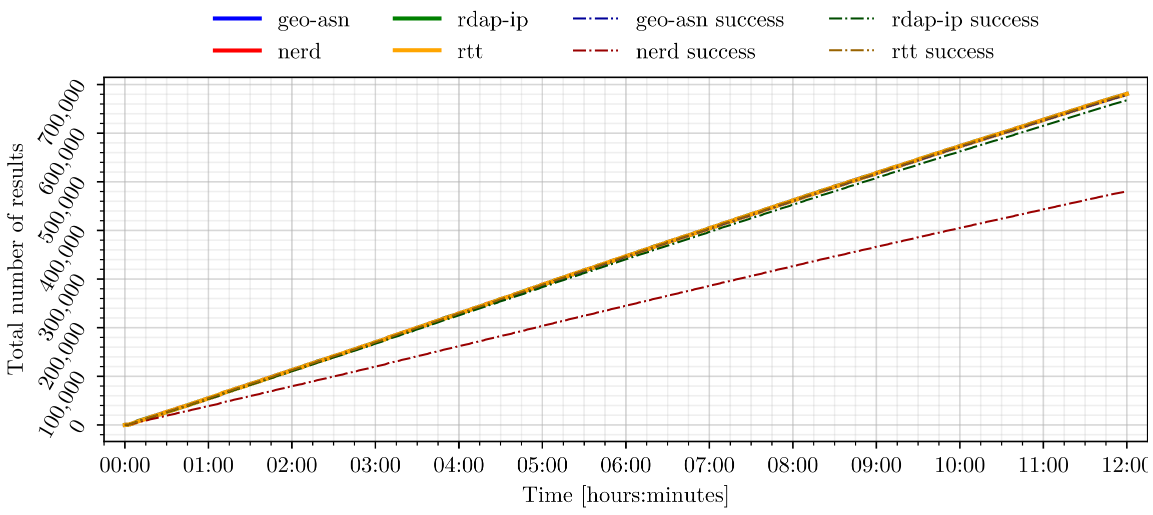


Figure H.28: Number of responses over time in real-time experiment #4.

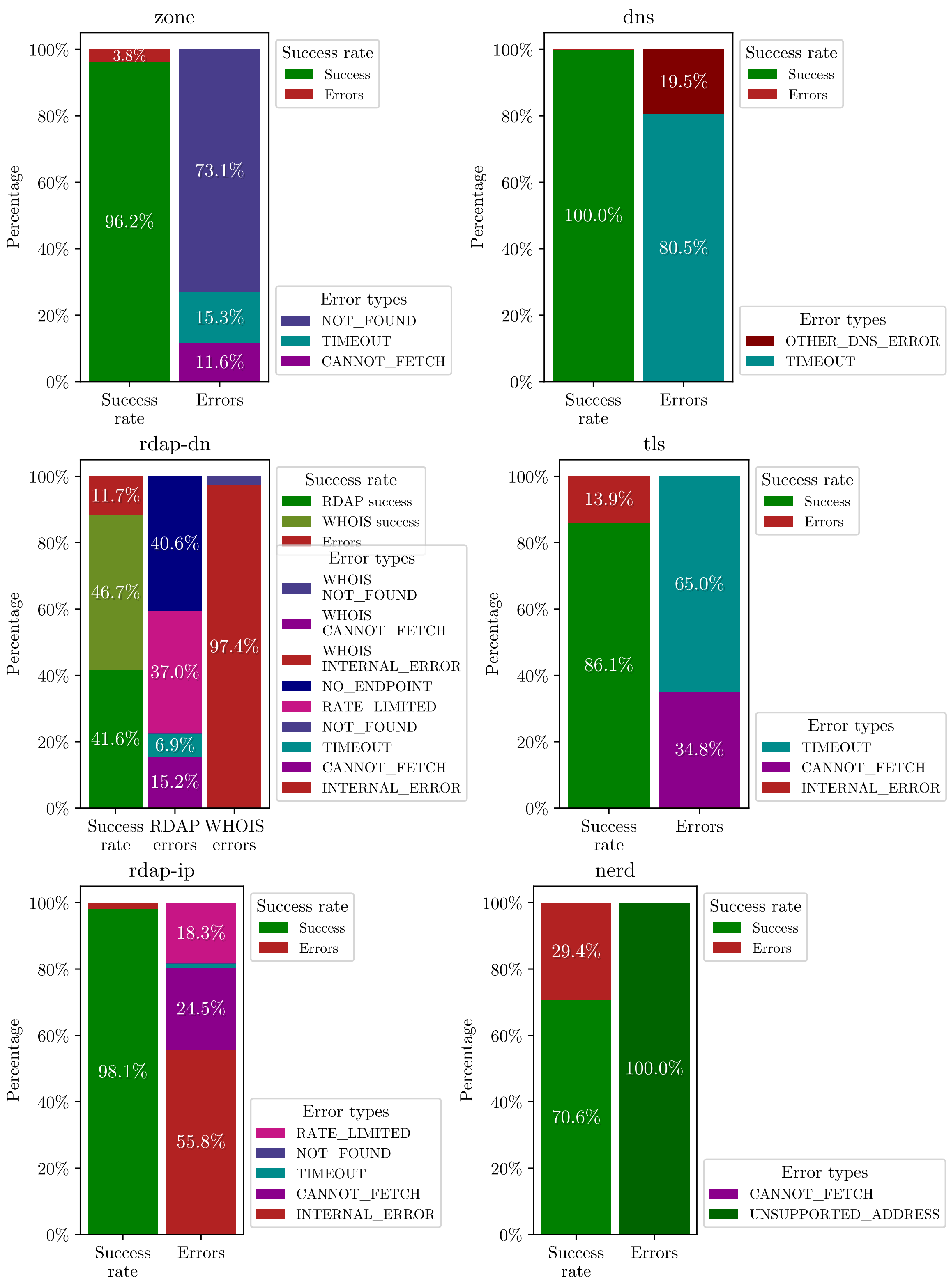


Figure H.29: Error types in real-time experiment #4.

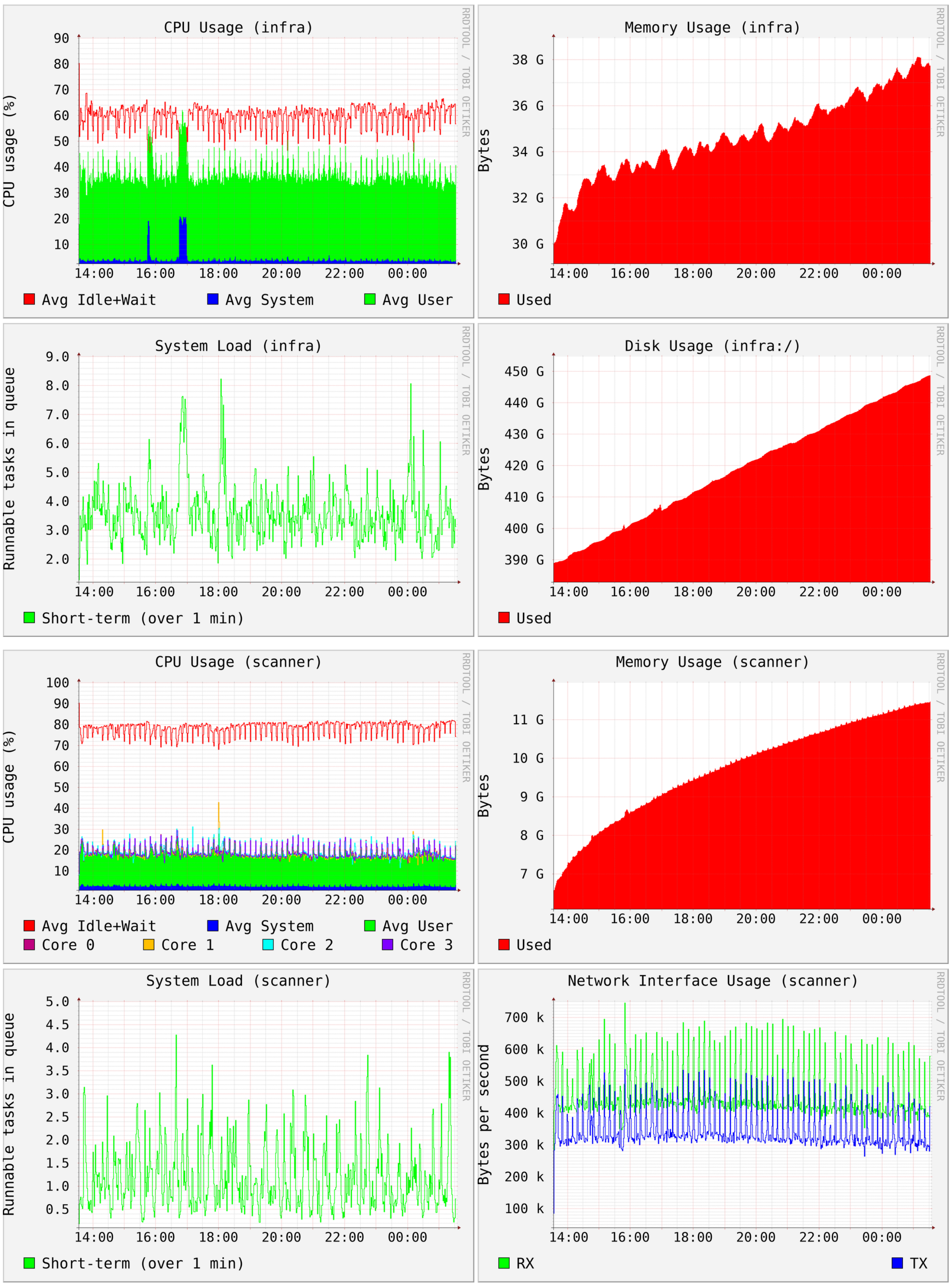


Figure H.30: System resources usage of the two VMs in real-time experiment #4.

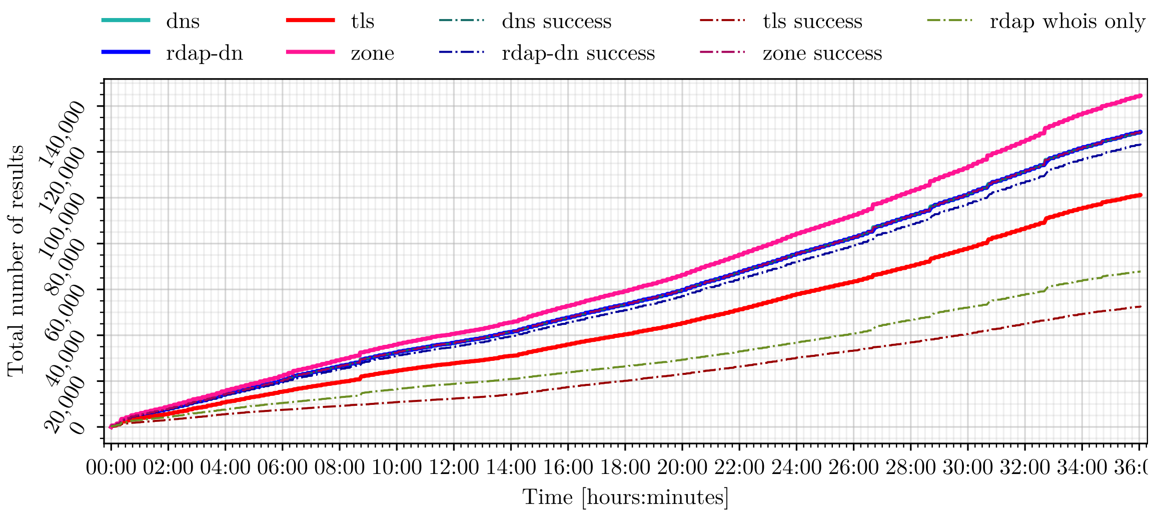
Real-Time Experiment #5

Collector	Tot [h:m]	AvgTput [req/s]	AvgColT [ms/req]	MnQdColT [s/req]
Zone	36:02	1.11	897	-
DNS	36:02	0.99	1008	10.54
RDAP-DN	36:02	0.99	1007	14.54
TLS	36:02	0.78	1281	3.81
GEO-ASN	36:02	1.36	736	0.12
NERD	36:02	1.36	736	0.13
RDAP-IP	36:02	1.36	736	0.64
RTT	36:02	1.36	736	6.84

Table H.13: Per-collector metrics in real-time experiment #5.

		Zone	DNS	TLS	R-DN	*WHOIS*	GEO	NERD	R-IP
NotFound	req%	6.86	0	0	0.03	0.44	1.87	20.08	0.00
Remote	req%	4.07	0.16	47.77	23.84	0.01	0	0.00	2.54
Internal	req%	0	0	0.33	0.10	6.97	0	0.00	2.38
RateLimit	req%	-	-	-	15.93	0	-	-	0.02
NoEndpt	req%	-	-	-	17.02	0	-	-	0
Other	req%	0.04	0	0	0	0	0	0	0
Total	req%	10.97	0.16	48.10	56.93	7.42	1.87	20.09	4.93

Table H.14: Error rates in real-time experiment #5.



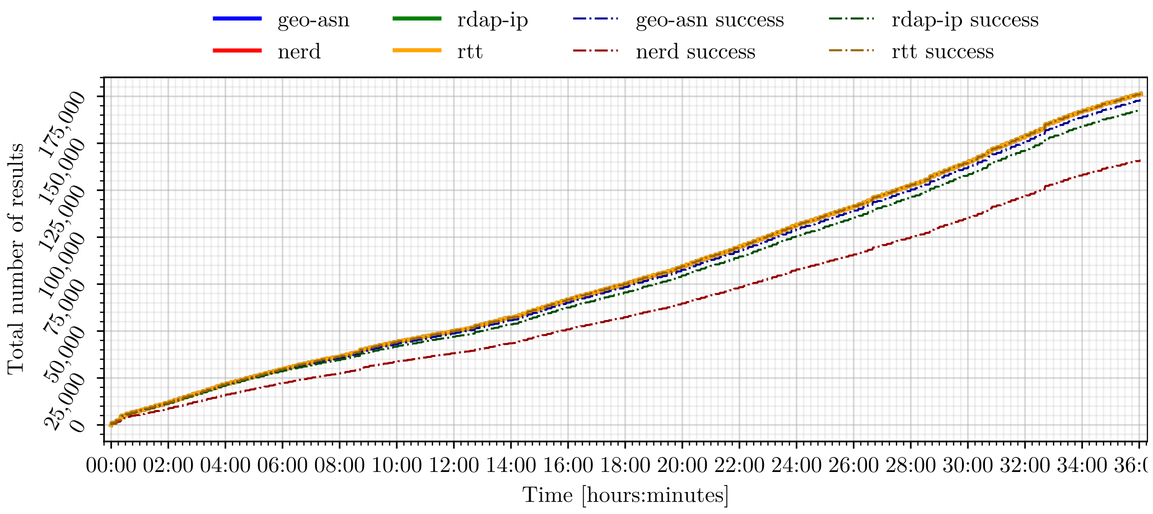


Figure H.31: Number of responses over time in real-time experiment #5.

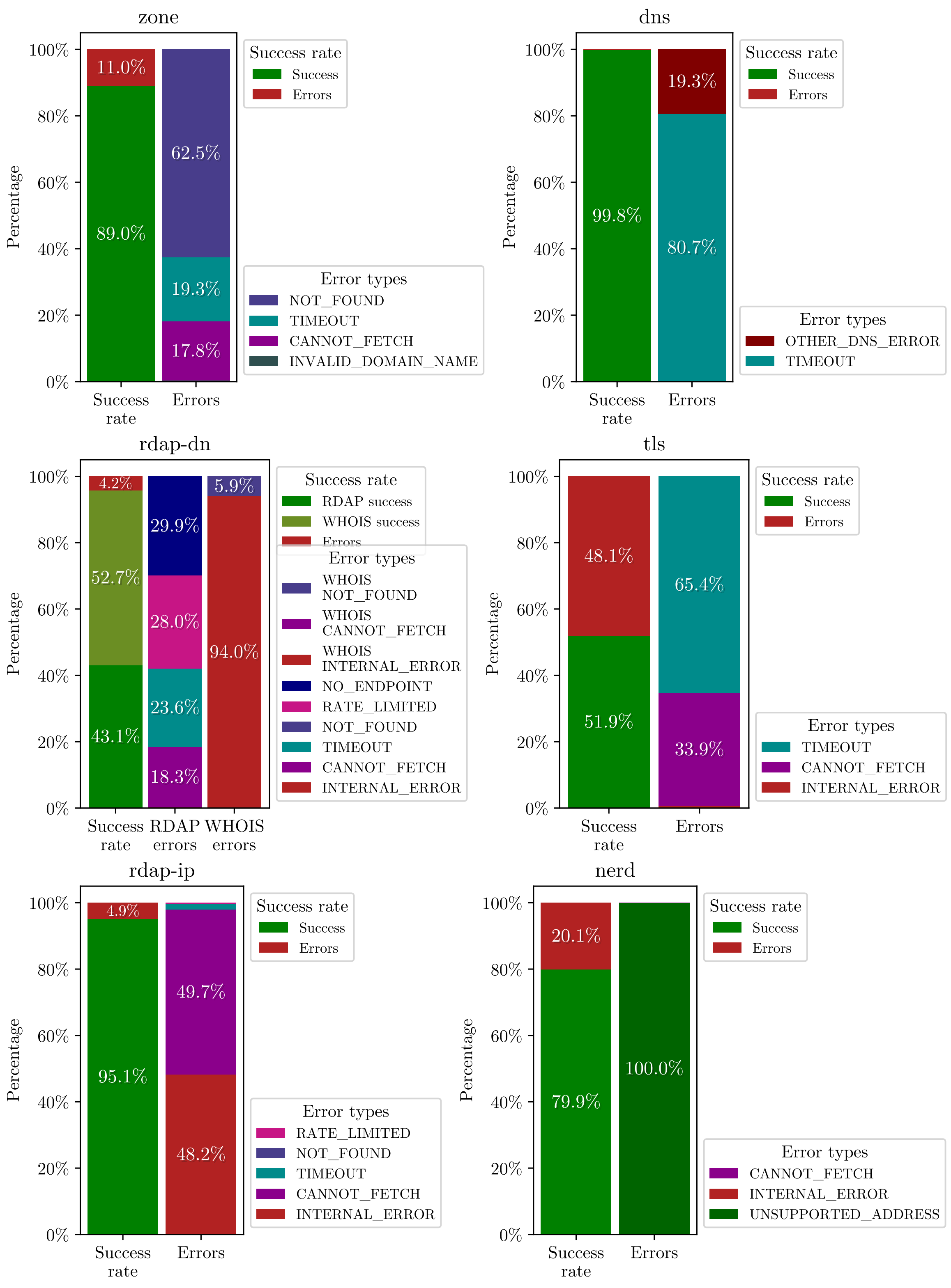


Figure H.32: Error types in real-time experiment #5.

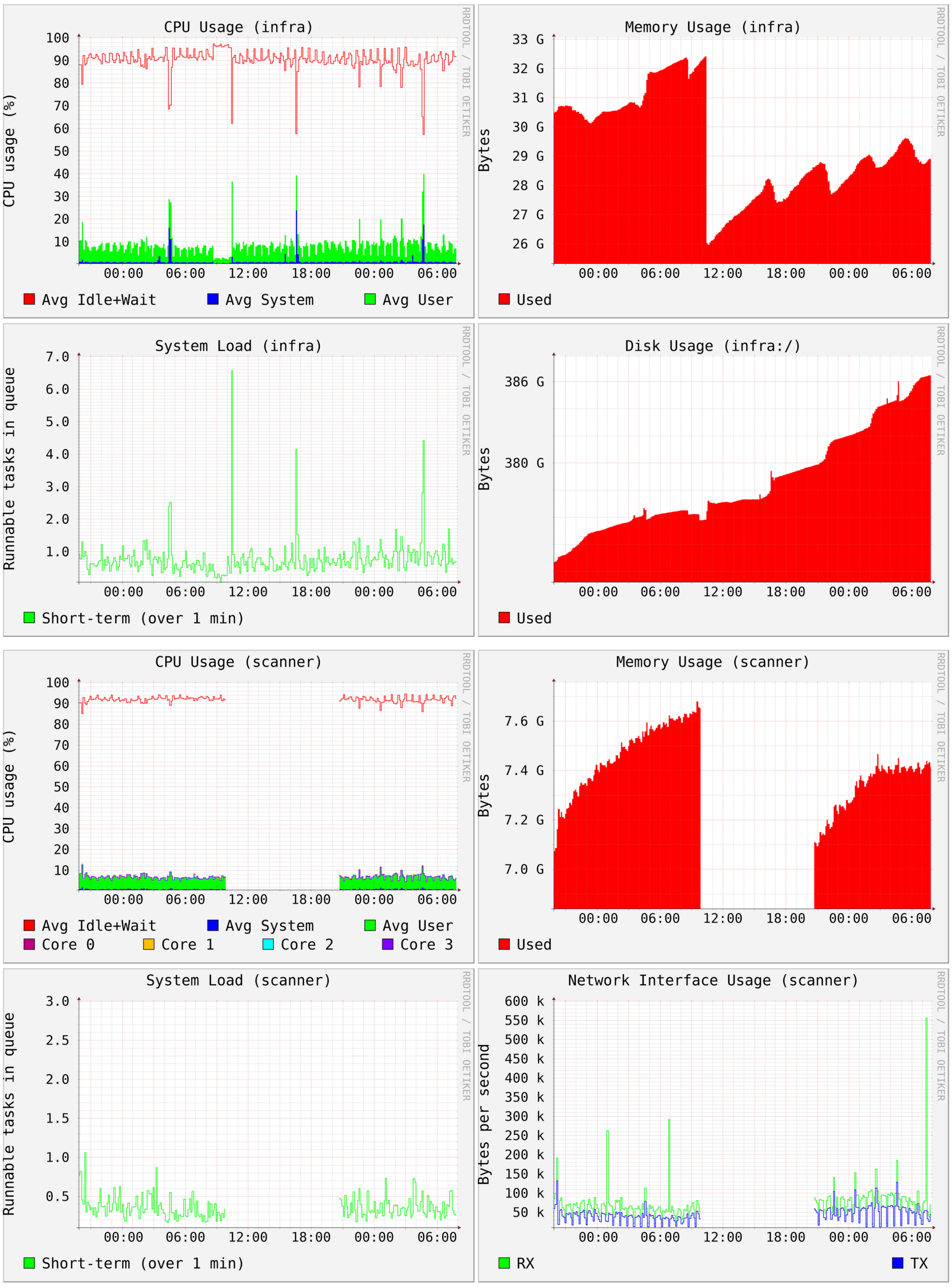


Figure H.33: System resources usage of the two VMs in experiment #5. The measurements are missing for the *scanner* VM between 10:00 and 21:00 because of an unexpected system restart killing the collectd measurement daemon.

Appendix I

Contents of the Attached Data Storage

/

- `analysis`: the scripts used for analysing the collection results
- `src`: the source codes
  - `java_pipeline`: Java-based pipeline components
    - `common`: models, shared constants
    - `connect`: transformations and converters for Kafka Connect
    - `serialization`: shared serialisation code
    - `standalone-collectors`: Parallel Consumer based collectors
    - `streams-components`: the data merger component
  - `python_pipeline`: Python-based pipeline components
    - `collector`: the collectors
      - `collectors`
        - `dns`: the DNS collector
        - `rdap_dn`: the RDAP-DN collector
        - `rdap_ip`: the RDAP-IP collector
        - `rtt`: the RTT collector
        - `zone`: the zone collector
    - `common`: models, shared constants, logging and utility functions
    - `extractor`: the feature extractor

- `config_manager`: the configuration manager
- `standalone_input`: the standalone input controller

- `infra`: the infrastructure Docker Compose
  - `client_properties`: configurations for the components
  - `connect_plugins`: plugins for Kafka Connect
  - `connect_properties`: configurations for Kafka Connect
  - `data`: runtime files for the collectors
  - `db`: database init scripts and configurations
  - `dockerfiles`: Dockerfiles for the supplementary services
  - `envs`: environment files for the services
  - `extractor_data`: supplementary files for the feature extractor
  - `geoip_data`: the GeoLite2 databases (by MaxMind)
  - `kafka_scripts`: Kafka setup scripts
  - `misc`: testing data and other supplementary files
  - `mongo_aggregations`: example MongoDB aggregations