triangulum

DEMONSTRATE·DISSEMINATE·REPLICATE

UiS Module Implementation Report

D2.2 Cloud data platform

and

Subtask 5.4.2 Data analytics toolkit

February, 2018 (M37)

Project Coordination:
Fraunhofer Institute IAO

H2020-SCC-2014-2015/H2020-SCC-2014:
'Smart Cities and Communities solutions integrating energy, transport, ICT sectors through lighthouse (large scale demonstration - first of the kind) projects'

Collaborative Project — **GRANT AGREEMENT No. 646578**

Contents

Executive Summary 5

Revision History 6

Conventions 7

1 Introduction 10
1.1 Motivation and objectives . . . 10
1.2 Context and orientation . . . 11
1.2.1 Project-specific conditions . . . 11
1.3 Report overview . . . 12
1.3.1 Report structure . . . 12
1.3.2 Who should read this report? . . . 13

2 Review of literature and related works 14
2.1 Distributed computing . . . 14
2.1.1 Fault Tolerance . . . 15
2.1.2 Distributed System Model . . . 16
2.1.3 Quorums . . . 17
2.2 ICT in other smart city projects . . . 17
2.2.1 MK:Smart . . . 17
2.2.2 Oulo smart city traffic pilot . . . 18
2.2.3 CityPulse . . . 18
2.3 Other relevant projects . . . 19
2.3.1 CKAN . . . 19
2.3.2 FIWARE . . . 19
2.3.3 New York Times and Apache Kafka . . . 20
2.4 Review Summary . . . 21

3 Architecture 23
3.1 Architecture Roadmap . . . 23
3.1.1 Purpose and Scope . . . 23
3.1.2 Organization . . . 23
3.1.3 Stakeholder Representation . . . 24
3.1.4 Viewpoint Definitions . . . 24

3.1.4.1 Viewpoint defintion: Implementation units decomposition viewpoint . . . 24
3.2 Architecture Background . . . 26
3.2.1 Problem Background . . . 26
3.2.1.1 System Overview . . . 27
3.2.1.2 Goals and Context . . . 28
3.2.1.3 Significant Driving Requirements . . . 28
3.2.2 Solution Background . . . 29
3.2.2.1 Architectural Approaches . . . 29
3.2.2.2 Requirements Coverage . . . 30
3.3 View . . . 30
3.3.1 Implementation Units Decomposition View . . . 30
3.3.1.1 View Description . . . 30
3.3.1.2 Primary presentation . . . 30
3.3.1.3 Context diagram . . . 30
3.4 Architecture Summary . . . 32

4 Design **33**
4.1 Design overview . . . 33
4.2 Data Acquisition . . . 34
4.2.1 Adaptors . . . 34
4.2.2 Queueing and Load Balancing . . . 34
4.3 Data Ingestion . . . 36
4.4 Data Storage . . . 36
4.5 Data Access . . . 37
4.5.1 Internal Access . . . 37
4.5.2 External Access . . . 38
4.6 Data Processing . . . 38
4.6.1 Exploratory Data Analysis Tools . . . 39
4.6.2 Batch Processing Tools . . . 39
4.6.3 Modelling . . . 39
4.7 Design Summary . . . 40

5 Implementation **41**
5.1 Pipeline and Components Overview . . . 41
5.2 Implementation division 1: Data collection framework . . . 44
5.2.1 Data intake form . . . 44
5.2.2 Adaptors . . . 44
5.2.2.1 CDPAdaptor . . . 45
5.2.3 Queueing and load balancing . . . 51
5.2.3.1 Parameters affecting fault tolerance . . . 52
5.2.3.2 Logstash parameters relating to Kafka . . . 53
5.2.3.3 Load balancing the publishing adaptors . . . 54
5.2.3.4 Load balancing the subscribing Logstash nodes . . . 54
5.2.4 Data Ingestion . . . 54
5.2.4.1 Sanitizing . . . 55

5.2.4.2 Filtering . . . 55
5.2.5 Data Storage . . . 55
5.2.5.1 Indexing . . . 55
5.2.5.2 Object Storage . . . 56
5.2.5.3 Monitoring . . . 56
5.2.6 Data Access . . . 57
5.3 Implementation division 2: Data processing framework . . . 57
5.3.1 Exploratory Data Analysis Tools . . . 57
5.3.2 Batch Processing and Modelling Tools . . . 58
5.4 Implementation Summary . . . 59

6 Deployment 60
6.1 Overview . . . 60
6.2 Methodology and technology . . . 61
6.2.1 Methodology . . . 61
6.2.2 Technologies . . . 61
6.3 Local virtual machine environment with Vagrant and Virtualbox . . . 62
6.3.1 Deployment to single VM locally . . . 62
6.3.1.1 Key files . . . 63
6.3.1.2 Operations to enact the deployment . . . 66
6.3.1.3 Notes on context . . . 66
6.3.2 Deployment to multiple VMs locally . . . 66
6.3.2.1 Key files . . . 67
6.3.2.2 Operations to enact the deployment . . . 70
6.3.2.3 Notes on context . . . 71
6.4 Deployments to cloud (Cloud virtual machine environment with OpenStack) 71
6.4.1 Deployment to single VM on cloud . . . 71
6.4.1.1 Key files . . . 72
6.4.1.2 Operations to enact the deployment . . . 72
6.4.1.3 Notes on context . . . 76
6.4.2 Deployment to multiple VMs on cloud . . . 77
6.4.2.1 Key files . . . 77
6.4.2.2 Operations to enact the deployment . . . 78
6.4.2.3 Notes on context . . . 79
6.5 Alternative deployment schemes . . . 80
6.5.1 Ideas for possible alternative deployment schemes . . . 81

7 Replication Guide 82
7.1 Overview . . . 82
7.2 Demo implementation with vagrant and single node . . . 82
7.3 Distributed on hardware . . . 83
7.4 Cloud hosted . . . 83
7.5 New data sources, new adaptors . . . 84

8 Data analytics use case: Traffic flow analysis 85
8.1 Implementation of the graph 86
8.2 Technical steps in preparing the analysis 87
8.2.1 Transformations on time-related DataFrame columns 88
8.3 Defining and running an algorithm to find the vertices and edges of the subgraph 88
8.4 Summary of Analytics Use Case 89

9 Conclusion 91

Glossary 93

Acronyms 94

Bibliography 94

Executive Summary

University of Stavanger (UiS) has developed and implemented two information and communications technology (ICT) modules that enable cloud services for big data from smart cities. This module implementation was undertaken in the context of the Horizon 2020 project Triangulum, funded by the European Union (EU) (Grant Agreement number: 646578). The module implementation was adapted to the situational requirements of the project and its consortium of partners. At the same time, the UiS modules were implemented according to requirements mentioned in pages $18 - 20$ of the project grant agreement mandate , to harness modern ICT within a big data paradigm.

Furthermore, in accordance with the overall intentions of Triangulum, to develop in the Lighthouse Cities technological solutions that can be taken up by the Follower Cities, the UiS module implementation was undertaken with a strong emphasis on open-source technologies as the components of the software infrastructure system that the module implementation represents.

This report is intended as a technical document for the deliverable D2.2 to support future replication and further development of the module implementation and so keeps being updated. Some chapters provide higher-level perspectives, intended for a less technical audience. For more detail on who should read different parts of this report, see Sec. 1.3.2.

Revision History

Initial report, version 1.01.

2nd February 2018: First version of the UiS module implementation technical report in the Triangulum project.

Initial report, version 1.00.

30th January 2018: First version of the UiS module implementation technical report in the Triangulum project.

Written by Trond Linjordet (UiS), with contributions from Aryan TaheriMonfared (UiS), Julian Minde (UiS), Russel Wolf (UiS), Aida Mehdipourpirbazari (UiS), Faraz Barzideh (UiS), Mina Farmanbar (UiS), Nejm Saadallah (International Research Institute of Stavanger (IRIS)), and Rui Esteves (IRIS).

Conventions

This report uses the following conventions:

Glossary terms

The first time a term with a glossary entry is used in this report, the term is displayed in a san serif font. For example, a sentence including such formatting of a special term indicates that "special term" can be found with a brief description in the Glossary of this report.

Listings

This report contains *listings* such as this, where the contents of a program or script are given, or the commands used in the command-line interface (CLI):

Listing 0.0.1: Commands or program, `/file/path/filename`

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

In the body of the text and listing titles, references to parts of code, to CLI commands, file paths, or file names, are made by writing in the `typewriter` font.

Notation and terminology conventions

Naming conventions

The Triangulum modules are given both a three-digit *identifier*, and a name. The module names have been updated during the project to reflect growing clarity of purpose over the course of the development of modules, but the numeric module identifier can in many cases help to map the module to its original corresponding subtask.

The module identifier is constructed according to the following steps: The first digit,

X, is determined by the Lighthouse City, so that

$$X = \begin{cases} 3 & \text{if the module belongs to Manchester,} \\ 4 & \text{if the module belongs to Eindhoven, or} \\ 5 & \text{if the module belongs to Stavanger.} \end{cases}$$

The second digit, Y, is determined by the primary impact domain, so that

$$Y = \begin{cases} 2 & \text{if the module impacts Energy,} \\ 3 & \text{if the module impacts Mobility, or} \\ 4 & \text{if the module impacts ICT.} \end{cases}$$

Finally, the third digit, Z, is left to iterate up from 1, to easily distinguish modules from the same Lighthouse City with the same primary impact domain. Where possible, Z is chosen to match the numbering of the single corresponding subtask in the Grant Agreement. Likewise, the conventions for X and Y support this aim for matching with the Grant Agreement, while providing an explicitly systematic mapping.

Thus, the "topic" (used for managing the data, see Ch. 5) for a module-relevant data source being collected to the cloud data platform is constructed according to the pattern `mXYZ-datasource-subset`. Sometimes, this pattern is referenced, especially in Ch. 5, as `mXYZ-*`, for brevity.

For data sources that are not directly related to Triangulum modules, but which may be relevant to collect to provide a basis for comparison, or for other purposes, the following alternative pattern is suggested: `xXYZ-datasource-subset`. Here the lowercase `x` is left as literal, replacing `m`, while the capital X should follow the above convention as for modules.

The qualitative impacts of each module can be evidenced by impact indicators, which quantify the impacts based on module-relevant data. Typically (in practice exclusively), this means data generated as a matter of course in the module implementation or the operations of the module once implemented. Each impact indicator is given a numeric identifier, consisting of six digits, consisting of the pattern `XYZQQQ`, where QQQ iterates from 1 (i.e. 001) according to however many impact indicators can be defined for the module. Where a single impact indicator has multiple variants, the impact indicator identifier may be suffixed with a, b, c, etcetera.

Terminology within the report

This section discusses some of the terminology which may be unique or nuanced within the present report.

In the report, the software systems that were developed as concrete, operational prototypes are referred to as the module implementation. This refers to both the UiS modules in the Triangulum project.

The software systems in the module implementation consist to a large extent of configurations and interconnections of pre-existing technologies. These have been chosen to fulfill functional requirments identified in Ch. 3. The overall system is said to consist of components. These in turn may be operational as deployments to dedicated clusters that together serve some function within the larger system of the module implementation.

The pre-existing and externally developed technologies used in the present work are discussed in terms of their proper nouns, which are capitalized in the prose. For example, the report describes and discusses Elasticsearch, Python, and Systemd. However, code, file names, and file paths, even when referring to the technologies' proper names, are always typeset in the `typewriter` font.

Chapter 1

Introduction

1.1 Motivation and objectives

The implementation of modules at UiS, *Module 542 Data analytics toolkit* and *Module 544 Cloud data platform*, was motivated by a need for a standard ICT solution for documenting and analysing the impacts of all modules in the Triangulum project. An additional motivation was the opportunity for undertaking more advanced analyses of the data collected to uncover possible insights or applications.

ICTs are central to implementing a smart city. Everyday tools and appliances are now produced with electronics that provide built-in opportunities for connectivity and coordination. This Internet of Things (IoT) trend can only be expected to become more widespread going forward. These technologies produce data both of internal functions and to inform various types of users and external devices of relevant facts. The multitude of data-producing objects represent an untapped potential, in that each object may only represent a single element in a complex, dynamic system. For example, a single geolocation device may provide movements of that device over time, but the set of comparable geolocation devices may provide valuable insights in the underlying dynamics of the device population.

In order to capture such data to enable insightful analysis, a system is required that can correctly ingest, reliably store, and intelligently process the data. A cloud computing solution can address all these criteria. Furthermore, a cloud computing solution based on locally situated hardware may in principle enable greater security and control than outsourcing cloud computing solutions to overseas commercial vendors. Finally, an open-source, commodity hardware cloud computing solution lowers the economic threshold (i.e. financial cost) to adopt this solution among Follower Cities and others who wish to replicate the present work in part or in full. The commodity hardware and open-source approach also enable minimal cost while scaling the implementation to the needs of the replicating party.

1.2 Context and orientation

The UiS module implementation has been guided by certain orienting principles, as well as the practical context in which the module implementation has occurred.

The implementation has harnessed modern cloud computing, big data, and IoT paradigms to serve the Triangulum smart city project and its overarching goal of replicability by developing with open-source, community-supported software for scalable and modular instantiation on commodity hardware.

By choosing to focus on *NIX (Unix-like) technologies for components, the UiS module implementation simplified the task of finding technologies that are both open-source and community-supported, as well as likely compatible with each other and the underlying computing environment.

Scalability, modularity, and commodity hardware go together in that the aforementioned modern paradigms share the well-supported premise that the generation, storage, and manipulation of data today occur in ways that are distributed on multiple devices. Thus, the economical approach to big data — as opposed to building custom, dedicated supercomputers — is to distribute individual large computing tasks among multiple commodity-level computers. This is discussed further in Sec. 2.1.

In addition, the context of the module implementation includes the parallel implementation at UiS, outside Triangulum, of the appropriate hardware infrastructure to support the software infrastructure implementation discussed in this report. The modules were implemented on a cloud computing platform, the CIPSI Computing Platform (CCP), operating at the University of Stavanger data centre. CCP thus constitutes a major aspect of the context for the module implementation within Triangulum.

Nevertheless, the implemented modules are compatible with any equivalent alternative cloud computing platform that can provision the same type of resources.

The modules presented in this report were developed to an operational prototype, or proof-of-concept, level of maturity. Further development would be required for the system to be appropriate for an enterprise-scale production level of operation. Such further development may well be appropriate in the context of an open source collaboration on the systems presented in this report.

1.2.1 Project-specific conditions

Triangulum created some specific starting conditions that affected the implementation of UiS modules, and the progression of the project as a whole also affected the UiS modules in a number of ways.

First of all, the Module 544 Cloud data platform (Deliverable 2.2) was intended to support Work Package 2 (WP2) reporting, by collecting data regarding module implementations throughout the Triangulum consortium. Note that *modules* in the context of Triangulum refers to the specific technological solutions implemented as project deliverables by the various partners and linked third parties in the Triangulum consortium. Because of this, the parts of the UiS module implementation in this report are referred to as subsystems or components.

The data collected must be used to calculate quantities that in turn support WP2

reporting. Since WP2 undertakes to report on the qualitative impact of the various modules implemented in Triangulum, the function of Module 544 relative to WP2 is to both collect the supporting data and calculate from these the impact indicators that quantify the impact of the various modules in Triangulum. In effect, the systems implemented as the *Data processing framework* within the cloud data platform overlap with the systems implemented as Module 542.

Each data source encountered was unique, and required the development of a custom *adaptor* to connect the specific data source to the general *Data collection framework* of the cloud data platform. This work required intensive and iterative liaising with technical personnel at the relevant data provider.

Under the Grant Agreement, any partner or linked third-party is permitted to withhold data from the aformentioned collection, if this withholding is done to protect either the privacy of individuals, or due to proprietary commercial concerns on the part of the data provider.

The implementation was by necessity prioritized according to the data availability.

While the legalities and liabilities associated with data affected by concerns for privacy and propriety present a significant challenge in handling data across organizations, the UiS module implementation successfully negotiated these issues with respect to some data sources.

However, out of concern for the uncertainty regarding openness of some of these data sources, and due to limited server capacity on the part of the data providers, the full implementation details of the corresponding adaptors are omitted from this report to prevent inadvertent overload on those services.

1.3 Report overview

1.3.1 Report structure

Chapter 1 has introduced the objectives of the report and the module development that this report documents, as well as the situational context. Chapter 2 reviews information from relevant fields of inquiry, including relevant technologies considered for the module implementation. Chapter 3 describes the requirements of the system and the resulting architecture. Chapter 4 develops the design of the module implementation. Chapter 5 provides details on the implementation of the software infrastructure. Chapter 6 goes on to present the deployments of the implementation as increasingly complex iterations. Chapter 7 steps back to provide a practical overview of how the module implementation can be deployed by replicating Follower Cities, or other parties wishing to replicate the deployments or further develop the module implementation. Chapter 8 discusses the use case of exploratory data analysis undertaken on bus data towards modelling local traffic and developing a bus arrival time prognosis service. Finally, Ch. 9 concludes the report with an overall discussion of the module implementation and possible future directions.

1.3.2 Who should read this report?

This report is written with the expectation that several categories of readers may be interested. While the previous section (Sec. 1.3.1) gives an overview of the topics discussed, the organization of the report merits a further explanation to support the readership. Business people, administrative personnel, and laymen may find most of the chapters informative and interesting based on the topics as explained in Sec. 1.3.1. However, Ch. 5, Ch. 6, and Ch. 8 are written with an expectation that the readership has some familiarity with Linux, Bash (Bourne-Again Shell), and Python, as well as a basic understanding of computer science. This expectation reflects the necessary level of technical expertise to meaningfully engage with the topics described. Chapter 7 contains step-by-step instructions that may be used with a less rigorous requirement regarding technical background, although some is certainly still expected.

Chapter 2

Review of literature and related works

This chapter provides background for the module implementation presented in this report, beyond the Triangulum project. The background will provide a basis for comparison, and for the reasoning about the UiS module implementation presented in subsequent chapters.

This background includes a cursory overview of the topic distributed computing, a highly relevant field of study within computer science, in Sec. 2.1. Furthermore, a non-exhaustive survey of selected projects with relevant ICT implementations, especially those related to scalable data infrastructures, is presented:

Smart city projects other than Triangulum with relevant ICT components are discussed in Sec. 2.2. Projects which are not smart city projects but have used relevant ICT components are described in Sec. 2.3.

The chapter concludes with a preliminary discussion of the survey and the implications for the UiS module implementation in Sec. 2.4.

2.1 Distributed computing

This section describes briefly the field of distributed computing, which concerns itself with distributed systems. This is highly relevant to the UiS module implementation, since the modules are all intended to reflect the underlying paradigms of big data and cloud computing, which typically rely on technologies based on distributed computing. The IoT paradigm aligns with the aforementioned paradigms, and hence distributed computing, in that IoT devices can generate big data that are typically managed with cloud computing.

A distributed system can be defined [1] as a collection of processes that collaborate in order to offer a certain service to another system or end-user. The collaboration is realized by so-called distributed algorithms. These algorithms aim at solving the agreement problem with respect to distributed processes. Distributed algorithms therefore assume computational environments that enable operations on concurrency, which requires — among other attributes — synchronization and consistency. Since the individual processes that make up the distributed collaborative process will realistically fail with some frequency, distributed algorithms are designed to handle process failures with some strategy. Such strategies must assume a failure model, but different distributed algorithms may assume

different failure models. In practice, failures models are typically stochastic.

Distributed algorithms are devised to satisfy two main classes of required properties: *safety* and *liveness* properties. A safety property is a property that must never be violated during the distributed process. A liveness property is a property that the processes running the distributed algorithm must always be able to eventually achieve. [1]

An algorithm property is said to be *safe* if is guaranteed by the algorithm that the distributed process will never enter a violated state with respect to that property. An algorithm property is said to be *live* if the algorithms guarantees that the distributed process will always advance the system toward a desired state with respect to that property.

For example, the Paxos algorithm [2] is aimed to solve the agreement problem and guarantees safety, but not liveness with respect to the state (agreement) that Paxos is intended to achieve.

2.1.1 Fault Tolerance

Fault tolerance, as the term suggests, is the ability of a system to tolerate faults, or failures. A fault or failure can be defined as the event that a process deviates from the algorithm. [1] In distributed computing, fault tolerance means that individual processes can fail while the overall distributed process continues to operate correctly to some acceptable extent. The expectations about what constitutes an acceptable level of functioning after some failure have been developed within the topic dependable computing.

Dependable computing addresses the following major system properties in pursuit of fault tolerance [3]:

- **Availability:** Availability is a property that defines the ability of a system to perform its function whenever needed. Availability is typically quantified by the probability that the system under consideration is available at any particular point in time.

- **Reliability:** Reliability is a property that defines the ability of a system to continuously perform its function during a time interval. Reliability is typically quantified by the mean interval between failures.

- **Safety:** Safety in its general sense, is the ability of a system to guaranty that nothing wrong will happen. Safety is a property that is much needed in Critical Systems [4], such as nuclear plants, or high pressure control process plants. Fail-safe mechanisms exemplify means that ensure this property. Note that this definition does not contradict the mathematical definition of Safety discussed above, which is a guarantee that a system will never enter an undesired state.

- **Maintainability:** Maintainability is a property that defines the ability to repair, upgrade, or even modify a system without compromising Safety, Availability and Reliability. This is clearly very challenging, but also a highly desirable property. Maintainability enables a system to continue running while being modified.

- **Recoverability (resilience):** Recoverability is the ability of the the system to return to a functional and operational state after failure.

2.1.2 Distributed System Model

Note that a distributed algorithm not only assumes a failure model, but also a distributed system model, which underlies and affects the design of the algorithm itself [1]. Influential aspects of the distributed system model include assumptions about reliability of links between processes, system synchrony, failure severity, and stochasticity (or not) [1].

A distributed system model constitutes a framework for studying and developing distributed systems. The framework consists of three abstractions: a process abstraction, a communication abstraction, and a failure detection abstraction.

A process abstraction defines the unit that performs a computation as described by a distributed algorithm.

A communication abstraction (or link abstraction) is used to represent the network component of the distributed system.

Distributed algorithms may rely on an ability to detect failures among the constituent processes. This is addressed in the distributed system model by a form of failure detection abstraction. Because it is almost impossible to distinguish between process failure and communication failure [1], most practical distributed algorithms assume no communication failures and attempt to determine whether a process has failed. Under the assumption that communication failure does not occur, a failing process can be detected based on expected behavior. This expectation is a reflection of the failure models assumed by the distributed algorithm.

One failure model abstraction is to consider five levels of failure models with their respective simplified failure definitions [1]:

1. **Crash:** A process stops executing steps of the distributed algorithm.
2. **Omission:** A process fails to communicate appropriately with another process.
3. **Crash with recovery:** A process capable of recovering itself after a crash fails to do so appropriately, by either **(a)** not recovering a crash, or **(b)** looping infinitely through a cycle of crash and recovery.
4. **Eavesdropping:** Adverserial processes outside the distributed system inappropriately gain information about the internal processes.
5. **Arbitrary:** A process deviates from the distributed algorithm in any conceivable way. Such failures are also called *Byzantine faults* and encompass all unknown or unpredictable failures.

For example, a process that does not respond after a given timeout will be considered crashed. A more challenging problem is when a process does respond, but in a malicious manner. Here notions of identity (with cryptography), or byzantine fault tolerance become important.

The assumed process failure models make a significant difference between various distributed algorithms, with respect to their practical applications and their properties in terms of safety and liveness. The strongest class of distributed algorithm would be fault tolerant with respect to Byzantine failures.

2.1.3 Quorums

A process executes a computation given by a distributed algorithm and can fail in different ways as discussed above.

A satisfactory distributed algorithm always reaches distributed agreement, and is never fooled by faulty processes. Fooling in this context means assuming distributed agreement when in fact distributed agreement is not reached.

The main tool for designing distributed algorithms is called *quorum*. A quorum is a set of processes that hold a sufficient majority of the distributed system with respect to some task or property. In particular, a quorum is important because distributed algorithms rely on quorums to achieve distributed agreement. If f is the maximum number of faulty processes that the system can tolerate and N is the total number of processes (including the faulty process), the number of processes required for a quorum can be defined based on the failure model of the distributed algorithm.

For a crash fault-tolerant distributed algorithm, a quorum is a set of $\frac{N+1}{2}$ or more processes. Thus, if $f < \frac{N}{2}$, the distributed system is guaranteed at least one quorum.

In comparison, a Byzantine quorum is any set of $\frac{N+f+1}{2}$ or more processes. So in a quorum of $\frac{N+f+1}{2}$ there are up to $\frac{N+f+1}{2} - f$ correct processes (non-faulty processes). For Byzantine quorums, we furthermore have that : $N - f > \frac{N+f}{2} \rightarrow N > 3f$. For example, to tolerate 1 faulty process, a Byzantine fault-tolerant distributed system requires at least 3 processes. Likewise, to tolerate 2 faulty processes a distributed system would require at least 6 processes. [1]

This summary has provided a superficial overview of the type of theoretical considerations that affect the development of a computing system with distributed components, a category to which both the UiS module implementations belong.

2.2 ICT in other smart city projects

This section discusses ICT in smart city projects besides Triangulum, although the publicly available information on these projects is limited on a technical level.

A number of smart city projects have been undertaken around the world, and a number of these appear in principle to contend with ICT challenges that are similar to those of the UiS module implementation.

The following projects were deemed noteworthy in terms of their relevance to the UiS module implementation, and were investigated more closely, with details in their respective subsections: MK:Smart, Oulu Smart City pilot, and CityPulse.

2.2.1 MK:Smart

The city of Milton Keynes in the UK is developing a hub (`datahub.mksmart.org`), which includes data about energy and water consumption, transport data, data acquired through satellite technology, social and economic datasets, and crowdsourced data from social media or specialized applications.

Like the UiS module implementation in Triangulum, MK hub is planned to support the creation of analytics applications and services. The scope of Smart Data is wider in

comparison to Triangulum, as it emphasizes other areas (which in Triangulum would be called impact domains) such as water. MK: Smart Data used Open Digital Rights Language to represent and manipulate data policies, rules to represent permissions, prohibitions, and duties with respect to the use, re-use and re-distribution of datasets.

Based on its primary and affiliated websites, MK smart and its MK data hub appear to mainly collect and analyze static, historical data sets. The architecture of MK Data hub is described in detail by Mathieu d'Aquin et al. [5] Unfortunately, no detailed information about the specific technological choices is available, although CKAN is mentioned. See Sec. 2.3.1 for the evaluation of CKAN relative to the present development.

2.2.2 Oulo smart city traffic pilot

One smart city project that stands out as relevant to the Triangulum development in its scope and orientation is the Oulo Smart City pilot [6]. The Finnish city of Oulu developed a pilot that acquires, stores, and analyzes low-latency, real-time traffic data. To solve this specific use case, Oulu developed a horizontally scalable analytical platform optimized to quickly process large quantities of sensor data. Oulu's approach is fundamentally based on the Lambda architecture [7].

In Lambda, the raw data is delivered separately to batch and speed processing layers. The speed layer processes the most recent raw data in combination with the last results provided by the batch layer. The batch layer, which is slower and cannot provide low latency analysis, is optimized to process large volumes of historical data in the background. Oulu chose Apache Spark Streaming as the core solution for the speed processing layer, and RHadoop library and Hadoop MapReduce programming framework for the batch processing layer [6].

Spark is fast, since it can run in-memory distributed through a cluster. The ability to run in memory is an important advantage in use cases like Oulu that requires iterative computing tasks. Spark also runs and integrates well with Hadoop. Furthermore, this architecture relies on Flume as a message broker native to the Hadoop ecosystem, designed to deliver data to Hadoop HDFS. A detailed explanation of this pilot was presented by Altti Ilari Maarala, et al. [8].

Unlike Triangulum, this platform is specific to low-latency real-time sensor data. Thus, Oulu's architecture might not be flexible enough to accommodate a wider range of use cases.

2.2.3 CityPulse

The CityPulse project (`www.ict-citypulse.eu`) is a smart city project that aims to collect large-scale streaming IoT data. In principle, this is a highly relevant approach to smart city data. However, the CityPulse project delves deeply into semantic and ontological modelling of the data sources, which is beyond the scope of the Triangulum cloud data platform and its implementation, as explained in Sec. 2.4.

CityPulse has addressed the IoT and ICT challenges of a smart city in interesting ways, and has leveraged a larger, more focussed team than that which has been available to the UiS module implementation [9]. Nevertheless, lessons may be drawn from CityPulse for any

extensions of the present work, especially in cases where data sources have pre-established and well documented ontologies.

2.3 Other relevant projects

The UiS module implementation may also benefit from considerations from projects that focus on scalable ICT infrastructures which are not strictly smart city projects. This section discusses a few such projects.

2.3.1 CKAN

One software project that frequently appeared in the process of in reviewing and discussing smart city projects was CKAN (`www.ckan.org`), Comprehensive Knowledge Archive Network). CKAN is a free and open source solution designed and maintained by the Open Knowledge Foundation. It provides a repository and management system for open data. This includes configurable web publishing of data. CKAN is used by governments and several cities, such as Stavanger municipality (`open.stavanger.kommune.no`) and the Brussels-Capital Region (`opendatastore.brussels`). Similarly, the open data repository `datahub.io` is based on CKAN and hosts public and open datasets from 866 organizations.

CKAN has two different types of data storage: The Filestore and the Datastore. The Filestore stores whole files such as excel spreadsheets, pdf, txt documents, etcetera. These files are stored as Binary Large OBjects (BLOBs) and CKAN does not provide the means to access or query parts of that file. To access the contents inside of a BLOB, the user needs to download the whole file and open it with the appropriate software.

The DataStore is a general-purpose database with fine-grained access. Here, individual data elements are stored as separate records. The records are accessible and can be queried directly at the database with no need to download the whole dataset. However, this functionality requires that a data model and schema is known, and that the database is set up accordingly beforehand.

While the datastore can be used to store and access fine-grained data, it is not optimized for time series data, such as sensor IoT data produced in approximately continuous real time. For a more specific example relevant to Triangulum, this could include the real-time location of a bus, or the electricity usage of municipal buildings. Finally, CKAN is developed primarily to host and serve open data. While extensions to CKAN exist for applications with real time data (`extensions.ckan.org/extension/realtime/`) and for restricted access (`pypi.python.org/pypi/ckanext-restricted`), these extensions are not a core part of CKAN, nor well supported compared to the core CKAN project.

2.3.2 FIWARE

Another software project that was raised as possibly relevant was FIWARE (`www.fiware.org`), an open-source and free software ecosystem for smart applications development. The development of the platform was funded by FP7 EU projects (Project IDs 285248 and 632893, primarily) with a cumulative budget of over 100 million Euro, and this has been further extended by the Horizon 2020 program.

FIWARE is named after the concept of the future internet, which refers to the idea that the requirements placed on internet technology are evolving, and the research to address these evolving requirements. The ecosystem provides and promotes a variety of software components for various purposes, with the overall goal of connecting cloud computing and IoT and enabling innovation and development of new services with these components.

After reviewing several of the FIWARE software components, three major contraindications for the UiS module implementation have emerged:

1. The IoT integration appears most likely to facilitate data collection within an organization working with an end-to-end solution, i.e. where the cloud computing back-end and the IoT back-end are implemented together. For example, the context management model does not provide any clear value when collecting data from sensors and an IoT back-end implemented without compatibility with the FIWARE NGSI standard.

2. The FIWARE software components often use pre-existing open source software projects, where the advantage of using the FIWARE version is not clear.

3. The FIWARE software components are not as well documented, especially in practical terms, nor as well supported by the online community, as the type of pre-existing open software projects that often are used within the FIWARE components.

Since the module implementations among Triangulum partners are not required to follow the FIWARE NGSI standard, FIWARE did not appear to have any advantages over directly developing and implementing cloud solutions with the technically most appropriate and best supported open source software components for a given function.

2.3.3 New York Times and Apache Kafka

The New York Times (NYT) has more than 15 million published articles (`https://query.nytimes.com/search`) that must be available with low latency to a wide range of consumer services and applications, including website search, personalization services, feed generators, as well as front-end applications, like the website and native apps. In the past 20 years, the NYT approached this requirement by building APIs that support pipelining the published data to the different consumers. The different consumer clients, developed by different teams, would use distinct schemas and APIs, and this would also change over time. This was an effort-intensive approach, requiring the maintenance of systems to normalize content passed from different producers of content to different consumers of content. In addition, older content was more difficult to access [10].

To solve these challenges, as well as others, NYT adopted a log-based architecture centred on Apache Kafka (`https://kafka.apache.org/`) as a message broker. All content producers store each of the articles in Kafka as a message. Kafka stores all these messages in a permanent immutable ordered log. The multiple content consumers need only subscribe to Kafka. By individually processing the log, each consumer client creates its own data store representing a *materialized view*, comprising a derivative extraction from the complete history of events (published content) captured in the log. Further developments on a

particular consumer's materialized view can then be deployed as a completely new consumer that consumes the entire history of the Kafka log from the beginning.

Since each asset (unit of content) is stored as a separate message, the log is said to be normalized, in a similar meaning as when the term is applied to relational databases. All the assets can be referenced on a many-to-many basis, but normalization enforces that a referenced asset always precedes the referencing element in the ordering of the log. This ensures that each asset is internal consistent at that point in the history. However, consumers like Elasticsearch do not necessarily support many-to-many relationships between objects. Thus consumers may require denormalized data to construct the appropriate materialized view. The NYT provides such denormalized materialized views using the Kafka Streams client library, which can update and re-bundle all assets and their references from the normalized complete history log to a denormalized log with only the current versions of the published content assets.

While the Apache Kafka technology is powerful and relevant in several ways to the UiS module implementation, it is important to note that the data from other Triangulum modules to be collected at the cloud data platform is expected to represent segments of histories that, once recorded, do not require the kind of iterative updating process required by the NYT.

2.4 Review Summary

In light of the above survey of theoretical background and survey of smart city and other relevant ICT projects, a preliminary summary is warranted regarding what these other projects illuminate about the UiS module implementation:

Note that all the above smart city projects serve local requirements, whereas the UiS module implementations in Triangulum are intended to serve the project as a whole, with a primary mandate to support reporting and assessment.

These other smart city projects have different objectives and priorities than the Triangulum cloud data platform, but some are still potentially relevant. For one thing, the projects collect city data, and may approach their tasks with technologies that might be of use in the UiS module implementation. The projects have therefore garnered at least cursory scrutiny in the preparation of the present work.

It is impractical to apply a formal ontology engineering approach in the context of Triangulum because the set of data sources that could be connected to the data collection framework is not finite or limited to a small number of domains, and these data sources are not a priori assured to have well-defined, static data models. Furthermore, the data sources were often not developed prior to the development of the UiS module implementation. Many Triangulum modules had not been fully developed and implemented at the time when major design decisions had to be made regarding the data collection framework in the cloud data platform.

Hence, a comprehensive ontological treatment of the indefinite number of data sources is not tractable along with the other objectives and constraints of the UiS module implementation.

With many data storage solutions, such as CKAN, it is typically impractical and complicated to change the data schema in the middle of ongoing data collection processes.

In addition, the complete set of data schemas that must be accommodated cannot in general be known *a priori*. The UiS module implementation therefore had to be data model agnostic in its approach.

In particular, due to parallel development of the UiS module implementation and the module implementations that represent its data sources, the UiS module implementation was developed to be versatile enough to accommodate schemas of maximum variety and variability.

While the review of background and relevant work presented in this chapter, as well as the discussion of the attendant implications, is non-exhaustive, it is hoped that this provides an illustrative set of examples for the context of the UiS module implementation.

Chapter 3

Architecture

3.1 Architecture Roadmap

3.1.1 Purpose and Scope

This chapter describes the architecture of *the system*, i.e. the combined UiS module implementation in the Horizon 2020 Triangulum project. While grounded in software architecture standards such as ISO 42010:2010 (`www.iso-architecture.org/42010`) and in particular the *Views & Beyond* [11] approach, the present work takes a pragmatic approach and provides an abbreviated architecture in light of the fact that this chapter exists in the context of a report that aims to cover more than merely architecture.

This means that a single view of the architecture is presented in this chapter, reflecting the concerns of the developer stakeholder group. This means the architecture documentation primarily supports the implementation and maintenance of the system. Future versions of this report may expand this chapter to include additional views that reflect the concerns of other stakeholder groups, insofar as this has reasonable utilitarian value.

Note that (software) architecture here is defined as an abstract exposition of the software elements in the overall system, in terms of their relationships and the properties that affect how the elements interrelate. This excludes the internal details of each element's implementation. Note also that this chapter uses the term implementation unit, instead of module, in deference to the special meaning module has in Triangulum.

3.1.2 Organization

This chapter is divided into sections along the lines of typical software architecture documentation. Note that documenting the sequence of changes to this document is relegated to the front matter of this report, under Revision History on page 6.

Section 3.1 gives an overview of the chapter structure and what stakeholders' concerns are addressed. Section 3.2 gives an overview of the system, its background, and the rationale behind the architecture. Section 3.3 presents the implementation unit view of the system architecture.

3.1.3 Stakeholder Representation

This subsection lists the stakeholder roles considered in the development of this architecture, and their respective identified concerns.

The primary stakeholder roles identified are users, acquirers, developers, and maintainers. In the Triangulum context, developers and maintainers are largely the same persons, i.e. UiS researchers working on the module implementation, and their concerns are integrated in the perspective of continuous development. Users are also likely to be mainly UiS researchers, although it is hoped that technical personnel among other consortium members in Triangulum, both from academic and industry partners, will be users and acquirers. It is also hoped that there will be users and acquirers among technical personnel in the Follower Cities.

In general, it is expected that stakeholders' concerns reflect how the architecture can usefully serve their respective roles by providing a comprehensible high-level view of the system. More specifically, typical concerns that the architecture should address include:

- **System purpose:** What is (are) the purpose(s) of the system?
- **Suitability:** How suitable is the architecture support achieving the system's purpose(s)?
- **Feasibility:** How feasible is the construction of the system?
- **Risks and impacts:** What are the potential risks and impacts of the system to its stakeholders?
- **Maintainability/evolvability:** How can the system be maintained and be developed further?

Note that all these questions reflect relevant concerns, to some degree or another, for all the stakeholder roles identified. For example, users may be less concerned about the feasibility of construction in that users are not involved until the system is constructed and operational. These different degrees of relevance of each concern with respect to each stakeholder role is shown in Table 3.1. Each cell reflects how relevant the concern (column) is to the stakeholder role (row) by the following encoding:

– not very relevant,
+ somewhat relevant, and
++ very relevant.

3.1.4 Viewpoint Definitions

3.1.4.1 Viewpoint defintion: Implementation units decomposition viewpoint

3.1.4.1.1 Abstract

The *implementation units decomposition viewpoint* presents a system as a set of non-overlapping, hierarchically decomposable [12] implementation units. This viewpoint should produce a unique presentation once applied to any given system.

Table 3.1: Stakeholder concerns matrix

	System purpose	Suitability	Feasibility	Risks and impacts	Maintainability/evolvability
Users	++	+	–	+	+
Acquirers	++	+	+	++	++
Developers	++	++	++	+	+
Maintainers	++	++	+	+	++

3.1.4.1.2 Stakeholders and Their Concerns Addressed

This viewpoint primarily addresses the concern of system purpose and the suitability of the system to support that purpose. However, it is hoped that the viewpoint also informs feasibility and maintainability/evolvability concerns adequately, by showing the separation of functional concerns, which in turn enables the modularity of the implementation units. Finally, while risks and impacts are not explicitly addressed, the clarification of system purpose and suitability provide the primary background required for any future work on risks and impacts.

Given the stakeholder concerns matrix as presented in Table 3.1, the degree to which this viewpoint addresses each stakeholder is clear. All stakeholder roles have a majority of their concerns addressed by this viewpoint, although it may provide more information than necessary for users. However, risks and impacts, which are a concern to all stakeholders, are not explicitly addressed by this viewpoint.

3.1.4.1.3 Elements, Relations, Properties, and Constraints

The primary elements in this viewpoint are implementation units, which can be hierarchically decomposed, in which case an element can relate to its sub-elements as constituents. Each implementation unit element has a name and a functionality, and software-to-software interfaces with other software elements. The flow of information from humans and data sources is also included in this viewpoint, e.g. as directionality of software-to-software interfaces, to clarify system purpose and the suitability of the interrelated implementation units.

3.1.4.1.4 Consistency and completeness

A view is complete and consistent with this viewpoint when the latter is applied to a system such that:

1. Each element has at most one parent.
2. Each major functionality is provided by exactly one element.
3. The functionality of the the set of elements meets the system requirements.

In addition, the design of the system, discussed in Chapter 4, is consistent with this architecture viewpoint if all implementation units correspond to components of technologies in alignment with current procurement decisions.

Likewise, the implementation of the system, discussed in Chapter 5, is consistent with this architecture viewpoint if all source code can be correlated with specific implementation units.

3.1.4.1.5 Viewpoint Source

This viewpoint was adapted from the *Views & Beyond* [11] approach and associated documentation template.

3.2 Architecture Background

3.2.1 Problem Background

The constraints exerting significant influence over the architecture fall into two categories: Triangulum task/deliverable descriptions, and the supporting UiS infrastructure development, i.e. the CCP.

In the Triangulum Grant Agreement (i.e. EU Grant Agreement number 646578), the UiS deliverables are described in connection with Work Package 2 (WP2) and Work Package 5 (WP5). As well as reporting requirements, the UiS deliverables include the implementation of two modules. These are described in the Grant Agreement in various ways. The mandate these descriptions constitute can be summarized as follows:

Regarding Module 544 Cloud data platform (D2.2):

- A cloud data platform is to be developed to support the collection and storage of data generated by partners and WP2.
- The data sets to be collected by the cloud data platform relate to the "implementation of the technological solutions in each city", i.e. the Triangulum modules.
- The cloud data platform is to support "automated assessment and integration" of these data.
- The cloud data platform is to help document the impacts of the Triangulum modules.
- The cloud data platform is to "facilitate open access to the data from the cities and provide WP2 with a single point of reference for the impact assessments." Note that this point is confined to "as openly available as proprietary interests allow."

- The cloud data platform is to "enable key stakeholders and external users to interact with and use" the open data collected and stored.

Regarding Module 542 Data analytics toolkit (Subtask 5.4.2):

- The data analytics toolkit is to provide "a framework and generic tools for big data analytics."
- The data analytics toolkit is to enable the integration and analysis of the data collected and stored in the cloud data platform.

From these descriptions, it is clear that the two modules can most profitably be seen as parts of a single system, with Module 542 functionality dependent on and subsumed by Module 544 functionality.

The second category of major constraint is represented by specifications of the facility that can host the UiS module implementations when deployed and operating: CCP has been developed primarily by Dr. Aryan TaheriMonfared, as a necessary foundation in conjunction with the UiS module implementations with overt correspondences to the task/deliverable descriptions in Triangulum, and constitutes an infrastructure that can provide a variety of services for data storage and computation.

The physical and software infrastructure of the CCP data centre at UiS, can provision virtual machines (VMs) and network resources to host the UiS module implementation, including both the cloud data platform and data analytics toolkit. The allocation of the (virtualized) non-software environment is therefore largely given, and — to a lesser but still large extent — so is the runtime environment of the software once implemented and operational, since CCP must provide a finite number of VM variants (images specifying operating system; configurations of VM resources like memory and number of virtual CPUs (vCPUs)).

3.2.1.1 System Overview

Based on Sec. 3.2.1, an overview can be described of the planned integrated technical implementation of D2.2 and Subtask 5.4.2:

The two parts of the integrated implementation are the data collection framework and the data processing framework. The data collection framework is intended to contain the implementation units responsible for the acquisition and storage of data, as well as associated necessary secondary functionalities.The data processing framework is intended to contain the implementation units responsible for processing data, e.g. for analysis purposes.

Compared to the Triangulum task/deliverable descriptions, we can say that the data collection framework corresponds to the part of deliverable D2.2 that is not responsible for integrating or analyzing data, while the data processing framework corresponds to the internal deliverable Subtask 5.4.2. Thus, strictly the cloud data platform contains both the data collection framework and the data processing framework, and hence also the data analytics toolkit.

However, the data analytics toolkit is to be developed so it can be deployed independently of the data collection framework.

3.2.1.2 Goals and Context

The cloud data platform aims to collect, store, and process data from the other technological solutions (i.e. modules) implemented in Triangulum.

The architecture is intended to provide a high-level map of the required functionalities, represented by implementation units, while leaving specific technological choices as open as possible. This is intended to support more flexible implementation, maintainability, and future development, where technological choices are relegated to design (see Chaper 4). This segmentation of choices into discrete levels of abstraction is intended to support that future challenges to the system may be solved a lower levels unless contextual changes demand an escalation up the levels of abstraction to solve newly arisen problems on the level of design, or even architecture.

The life cycle of the UiS module implementation is not clarified beyond the project timeline of Triangulum. However, our aim when developing the module was that parts or all of the software will be replicated by Follower Cities and/or other cities, countries, etc.

3.2.1.3 Significant Driving Requirements

The driving requirements of the system architecture can be categorized as either qualitative or behavioral:

To collect and store data from a variety of external data sources — where neither the protocol or interface to set up data transfer, nor the data model or schema to store the data internally to the cloud data platform is known *a priori* — requires eventual specifications from data providers prior to data transfer, but also generalization and flexibility on the part of the data collection framework.

Likewise, the storage of data from the various data sources should be harmonized as much as possible to simplify the data processing of each source. This means the approach to develop the processing of one data set — especially extraction, transformation, and loading operations — should be as similar as possible. This means a very general approach to schema is required.

This means involvement is required from the technical personnel of the consortium member organizations that will act as data providers. This involvement is required insofar as each data source is unique and requires system and domain knowledge in order to set up ongoing data transfer from the data source into the cloud data platform. At the same time, the required involvement should be minimized to spare the efforts of both consortium member personnel and of UiS researchers.

Both for the sake of making the data collection framework separable from the rest of the cloud data platform as a data analytics toolkit, and to simplify future developments of the cloud data platform, the functionalities should be implemented in modular units in accordance with the principle of separation of concerns.

Since the big data paradigm is explicitly espoused in the task/deliverable descriptions, and the volume of data is not known a priori, the system and its implementation units should be independently horizontally scalable.

Likewise, the implementation units should be available to each other on an ongoing basis once the system operational, to support continuous data flow. Given the allocation environment CCP and the big data paradigm, the implementation units will be implemented

as modular subsystems that can be deployed in a distributed, fault-tolerant manner.

Thus, the primary quality attribute requirements of the system are:

- **Generality:** Supporting flexibility and low-effort requirements of human intervention to specify processes for each external data source.
- **Modularity:** Supporting maintainability/evolvability and scalability of the system and components.
- **Availability:** Supporting ongoing operations of interdependent functions.

Behaviorally, the requirements can be summarized as follows: The system must be able to

- Acquire data from external data sources, transferring the data in the "raw" format as it is made available.
- Ingest the data once acquired, meaning (1) load balancing pre-storage data flow as required, as well as (2) any required pre-storage processing of the data to harmonize it with the general storage solution.
- Store the data once ingested in a common, general way along with the data acquired and ingested from all the external data sources.
- Make the stored data accessible to the data processing framework.
- Extract the stored data via some access method and load the data in a way that is amenable to process.
- Process the data for the purposes of exploration, manipulation, analyses, and calculations.
- Store the results of processing, separate from but alongside the data originally collected from external data sources.

3.2.2 Solution Background

Having described the specific challenges of the system architecture in Sec. 3.2.1, the approach adopted to meet those challenges is developed presently.

3.2.2.1 Architectural Approaches

As a clear sequence of critical steps emerges from the system's intended functions, a pipeline motif is the natural underlying pattern for the system. In discussing the cloud data platform, it is therefore valuable to consider the primary direction of data flow. One can define the external data sources as upstream of the cloud data platform, and define persons or software clients receiving output from the data processing framework as downstream from the cloud data platform. Internal to the cloud data platform, the data collection framework is primarily upstream to the data processing framework, though the data processing framework should be capable of writing data to storage in the data collection framework, against the primary direction of the data flow.

3.2.2.2 Requirements Coverage

The qualitative requirements in Sec. 3.2.1.3 of modularity and generality are addressed by choosing open-source, community-supported software technologies for each implementation unit, with a focus on data model agnostic serialization for transfer between implementation units and schemaless storage.

The behavioral requirements are addressed by assigning separate implementation units (each a constituent of either the data collection framework or the data processing framework) to each of the required system behaviours. Note that access capabilities between programs are typically enabled directly by native application programming interfaces (APIs), or by custom connector programs that enable communication between APIs that are not natively compatible.

3.3 View

3.3.1 Implementation Units Decomposition View

3.3.1.1 View Description

This view reflects the implementation units of the system, in particular from the viewpoint of developers, with an emphasis on the movement of data (indicated by arrows) from external data sources, through the implementation units, and out to eventual external users and applications. Note that the view contextualizes the implementation units with external elements at the necessary points of contact.

3.3.1.2 Primary presentation

The view includes contextual elements, including the data provider, typically a Triangulum partner, or alternatively a linked third-party consortium member, and the data source, typically representing directly a Triangulum module at the data provider's disposal. Furthermore, external users and applications are contextual elements for this view.

The implementation units comprise the hierarchically decomposable elements of the system. The system as a whole is designated the cloud data platform, and contains two sub-elements, the data collection framework and the data processing framework. The data collection framework contains the implementation units data acquisition, data ingestion, data storage, and data access. Meanwhile, the data processing framework contains the implementation units exploratory data analysis, batch processing, and modelling.

Each of these low-level implementation units can be implemented as separate software components addressing the corresponding required functionality of the overall system. As required in design and implementation the software elements can be further decomposed in lower levels of abstraction.

3.3.1.3 Context diagram

This section describes the elements and relations shown in Fig. 3.1 to illustrate the implementation units decomposition view of the system that represents the architecture of

the UiS module implementation.

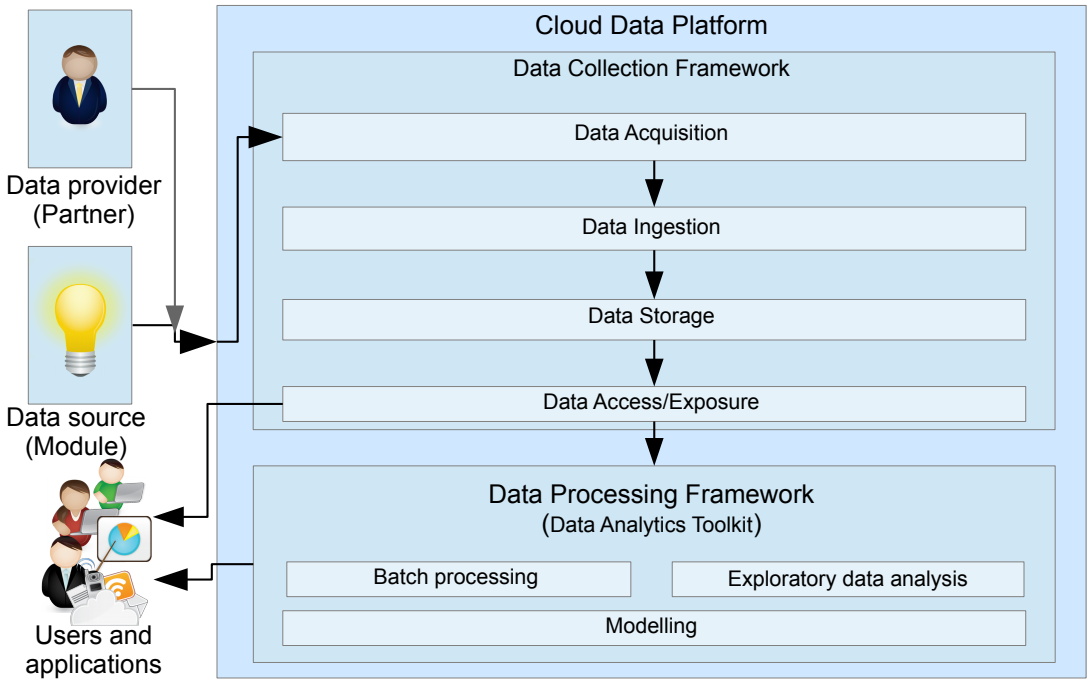

Figure 3.1: UiS module implementation architechture: Implementation units decomposition view.

Since the set of data sources that the cloud data platform is expected to collect data from is not amenable to rigorous ontological modelling given the project constraints, the data collection framework is designed to be data model agnostic. To support subsequent analysis, a simplified, human-readable data model is elicited together with technical personnel from the data providers in the data intake form. The need for a semantic data model for each data source was limited to a human-readable documentation that can aid human data analysts in constructing impact indicator calculations and analyses.

This is indicated in Fig. 3.1 by the upstream external human input along with the data coming from the data source itself. The process of ongoing data transfer from the data source to the data collection framework must be managed by an implementation unit called data acquisition. It is worth noting that this implementation unit must also bridge between the specific data source (of which there may in principle be an arbitrary number) and the general data collection framework.

The sequence of implementation units in the data collection framework should be clear within the context and abstraction level of this architecture. Note that data acquisition must meet specific requirements of individual data sources, which is challenging to generalize. However, assuming this requirement is met, the primary flow of data in the data collection framework should go in sequence through the implementation units of data acquisition,

data ingestion, data storage, and data access.

The data processing framework contains three implementation units, reflecting required functionalities with respect to exploratory data analysis, which is a largely *ad hoc* human endeavour with statistical and graphical tools, batch processing, which is the capability to process large amounts of (offline) data, and modelling, which here may be the combination of exploratory data analysis activities which can inform certain batch processing activities to build up a model that in turn can be used to respond to more local or recent data from a given source, for predictive purposes.

It should then be possible to give registered external users and services access to open data stored on the data collection framework. Likewise, registered external users should be able to access analytics tools on the data processing framework.

3.4 Architecture Summary

This chapter has provided an overview over the system architecture of the UiS module implementation. This overview included rationales involved in making the various system architecture decisions. The situational context informing the architecture has also been described in detail. The implementation units have been discussed in terms of their required functionalities and quality attributes. The essential relationships between the implementation units have also been indicated.

The system architecture may in principle be developed further, although the present form represents a practical distillation of the fundamental concepts involved in the module implementation. Future developments on the architecture documentation of the module implementation might include views on non-functional back-end requirements, and external user views.

Chapter 4

Design

4.1 Design overview

Having established the overall architecture of the UiS module implementation in Chapter 3, this chapter proceeds to discuss the design choices regarding each of the individual implementation units that constitute the architecture. The design represents the plan for how the architecture should be implemented. These design choices must address which technology can meet the quality and behaviour requirements, including critical relationships between the implementation units of the architecture.

Note the terminological distinction between an implementation unit, which is a subsystem defined on an architectural level of abstraction, and the software component, which is generally just one of many possible solutions to the requirements of the implementation unit. This chapter begins the planning of components that can meet the requirements of the implementation units. However, the distinction is made because other components could be designed and implemented based on the same architecture and implementation units.

This supports the quality attribute of modularity and the principle of *separation of concerns*, which in turn supports the evolvability of the system. For example, the system based on the same architecture could be updated by iterating through the implementation units of the architecture one at a time, and further developing the design and implementation of the corresponding component.

This chapter briefly goes through the design choices with regards to the components corresponding to each implementation unit, focussing primarily on choices of technologies that can meet the requirements. To properly contextualize the design choices for the UiS module implementation, note that the CCP is designed to provide Linux operating systems, which are open source, such as Ubuntu. For the sake of stability and broad online community support, Ubuntu 16.04 LTS (Xenial Xerus) (`www.ubuntu.com`) is chosen as the default operating system of the VMs hosting the UiS module implementation components.

The data collection framework is discussed in terms of data acquisition (Sec. 4.2), data ingestion (Sec. 4.3), data storage (Sec. 4.4), and data access (Sec. 4.5). Note that data access is discussed both in terms of internal access from the data processing framework to the data collection framework, and in terms of access to either framework by an external user or service. The data processing framework is discussed in terms of tools for exploratory

data analysis (Sec. 4.6.1), for batch processing (Sec. 4.6.2), and for modelling (Sec. 4.6.3).

4.2 Data Acquisition

The UiS module implementation in Triangulum must be able to collect data from a variety of external data sources that cannot be defined beforehand. This means the data collection framework must consist of a flexible and general set of components to handle data sources with diverse data models and formats, as well as unknown volume and velocity. Thus, it is clear that the component corresponding to the data acquisition implementation unit must consist of two types of subcomponents: a type that meets the specific requirements of an individual data source as it becomes known, and a type that can pass the data appropriately to the more general part of the data collection framework. These types of subcomponents can be called (a) adaptors, and (b) queueing and load balancing.

4.2.1 Adaptors

Since each data source may have not only a unique data model, but also specific methods for how data transfer should occur, some customization is required for each new data source that can be identified and specified with the help of technical personnel from the data provider. It must be assumed that each new data source requires a separate adaptor subcomponent to be developed that can undertake the transfer the data from the data source according to the particular protocol or interface that the data provider offers.

Some major interfaces (APIs) and protocols can be anticipated, but the specifics cannot be anticipated reliably. Once some data sources are characterized and ongoing data transfer is established, a generalized approach to implementing adaptors can be defined, but this aspect of the data acquisition implementation unit will require iterative and individual development efforts. Since the level and amount of required customization is anticipated to be high in the context of Triangulum, the appropriate approach is to use a scripting language.

Python is a versatile and popular programming language with broad support in the open source community that can be used for scripting a variety of custom functions. The Python programming language in all its release versions is open source. In addition, for Python many open source libraries exist that support functionalities beyond the core libraries in Python. Finally, since Python is more commonly used for data analysis purposes than otherwise similar languages such as Ruby, it is an expedient choice for scripting where necessary and amenable across the UiS module implementation. Thus, Python is the chosen default technology for general scripting purposes, and for data acquisition adaptors in particular.

4.2.2 Queueing and Load Balancing

The adaptors discussed in Sec. 4.2.1 are required to address the specific requirements to establish ongoing data transfer from individual external data sources into the general data collection framework. The adaptors are necessary to address the potentially unlimited variety of data models and interfaces/protocols that the external data sources represent

in a big data paradigm. However, the external data sources also represent a challenge in terms of the volume and velocity of data transfers.

The CCP context can set safety limits on these parameters, but to successfully collect data, the data acquisition stage should avoid overloading downstream components in case of bursty behaviour by data sources or adaptors. Thus, the queueing and load balancing component is downstream from adaptors. The role of this component is to buffer the data that is captured with various adaptors from their respective data sources, and to dispatch the data to the preprocessing component.

The chosen technology for queueing and load balancing is Apache Kafka, as it is highly flexible and has a proven track record in enterprise scale applications, as described in Sec. 2.3.3.

The communication mechanisms to implement this component follow the *message queueing* paradigm. In the cloud data platform, using a *message queueing* service for queueing and load-balancing decouples the adaptor components from the data ingestion components. This decoupling is achieved through a communication pattern called *publish-subscribe*.

In this pattern, one subsystem sends messages to another by means of a publish operation to an intermediary queueing service. Each message includes a topic and a message body. Any other subsystems that have performed a subscribe operation on the queueing service with a given topic as argument will then receive messages labelled with that same topic.

Since bursty behaviour in the data flow is a realistic risk upstream from this component, the queueing and load balancing component is required to be fault tolerant. This component attribute can be ensured by various means by the different candidate technologies. In the case of Kafka, fault tolerance is ensured by setting appropriate values for parameters such as number of partitions and replication factor in the implementation.

Note that Kafka cluster operations require a mechanism for consensus in among the Kafka nodes, which is solved by leader election. Leader election is a task used in many fault-tolerant distributed systems. The methods and mechanisms may vary. In the case of Kafka, a separate Zookeeper cluster can be used to handle leader election. The Zookeeper cluster uses so-called atomic broadcast primitives in avoiding a split-brain scenario where a cluster divides into subsets of nodes that are not in agreement. Consequently, leader election is also a process that affects fault-tolerance.

Kafka was originally developed by LinkedIn [13] and released as open source in 2011. The code is available under an Apache 2.0 license, which is approved by the Open Source Initiative. Apache Zookeeper is also available under the Apache 2.0 license.

Kafka uses an intuitive semantic model, and is very clear about its queuing model. The main innovation with Kafka lies in the immutability of published messages, which is achieved by a so-called *commit log*. In this model, data is published to Kafka under topics. The data is stored for configurable period of time, which makes it possible for clients to read earlier messages that are kept in the *commit log*. The availability of such a log means that a data consumer not only can recover in case of failure, but can also be designed to handle blocks of messages at once.

4.3 Data Ingestion

Having acquired the data from external data sources, and passed into the pipeline of the data collection framework, there is a need to perform ingestion of the acquired data into storage. This involves preprocessing the data insofar as necessary in the process of transferring data to the storage component(s).

Necessary transformations may include inserting, combining, or omitting certain data fields. The contents of each message (i.e. data point) could be parsed, and subsequent processing could be based on this, such as storing different categories of messages from a single data source in different data sets, or completely dropping certain types of messages. The data ingestion component could in principle be used to harmonize the schemas of comparable data sets by imposing mappings from the data source schema to some preferred schema for storage in the data collection framework.

Regardless of the specific operation or purpose of any such transformation, a data ingestion component enables preprocessing the data before storage, which can save a great amount of time and effort compared to manipulating and harmonizing data sets only after the act of collecting data, typically called *data munging*.

Logstash is a distributed computing technology for parsing, filtering, and transforming incoming data before writing the data to storage. Logstash also supports transferring the preprocessed data to several different types of data storage. In addition, it is a technology that is built to work particularly well with Elasticsearch (see Sec. 4.4), and Logstash is part of the Elastic stack (`www.elastic.co`). Finally, Logstash is available under an Apache 2.0 license. For these reasons, Logstash is the chosen technology for the data ingestion component of the data collection framework.

4.4 Data Storage

Data storage is crucial when collecting data that is to be used in any way subsequently. The manner in which the data will be used informs the choice of storage technologies. As previously pointed out, the underlying data model or schema is not known beforehand with respect to the external data sources from which the UiS module implementation is intended to collect data.

Thus, the choice of storage technology should be flexible — approximately schemaless. However, insofar as the data source provides data with schema, these schema should be preserved. This is especially true with the initial stages of establishing data collection from a given data source, before there has been developed any particular mapping to a schema that is more useful within the Triangulum context than the native schema of the data source.

Given the data model agnostic starting point and an orientation to support data analysis, and especially exploratory data analysis due to the inherent unfamiliarity of the data sources, a highly flexible data storage solution was sought in the UiS module implementation. The chosen technology for this data storage component is Elasticsearch, another part of the Elastic stack that is also available under an Apache 2.0 license.

In addition to being built to work with the other Elastic stack technologies that are

used in the components of the UiS module implementation (see Sec. 4.3 and Sec. 4.6.1), Elasticsearch has the advantage of supporting bidirectional interaction with Apache Hadoop jobs, and hence with Apache Spark (see Sec. 4.5.1 and Sec. 4.6). Elasticsearch is also deployable as a distributed system, supporting the fault tolerance and availability of the stored data. Finally, the underlying architecture of this technology provides data storage where each collected data point is a document indexed in a search engine.

Ceph (`ceph.com`) object store is a second technology for distributed data storage that is included in the UiS module implementation. This is a technology that is provided directly by the CCP as volumes independent from VMs, and which can be used for back-up of data alongside the Elasticsearch cluster of VMs. Ceph is available under a combination of different open source licenses, primarily LGPL2.1.

Ceph is an integral technology in CCP, and provides the VM-specific virtual disks via Ceph block storage, which underlies the Elasticsearch data storage. The data storage based on Elasticsearch is intrinsically temporary, based on the principle that the VMs that host the nodes in the Elasticsearch cluster themselves are temporary. In contrast, the Ceph object storage service is provided independent of data storage on VMs, for long-term archiving and back-up, for sharing of data, and for recording VM images. The Ceph object storage service is analogous to the S3 service provided by Amazon Web Services, and the interfaces are compatible.

4.5 Data Access

Once collected and stored, the data must be made available for processing. This is differentiated into the component categories of internal access, and external access. Internal and external access refer to whether the user or program trying to access the stored data is internal or external to the cloud data platform and its network. Note also that external access development has not been prioritized up to this point due to other overriding requirements. However, depending on priorities, the development of external access components can be undertaken as an immediate next step.

4.5.1 Internal Access

The components in the data processing framework can access the data storage components in a number of ways. Mainly, the API of Elasticsearch is used. In principle, the S3-compatible interface of Ceph could also be used. However, with Ceph object storage providing back-up primarily, this would not be the default approach unless specific advantages can be identified. This would likely be most relevant if batch processing (see Sec. 4.6.2) could be made more efficient by accessing data directly from the Ceph object storage.

For data storage access by exploratory data analysis components, the appropriate access interface depends on the component. See Sec. 4.6.1 for further details.

4.5.2 External Access

External access, meaning providing controlled access to the collected data by users or programs outside the CCP network, has yet to be developed in the UiS module implementation.

However, the external access component is currently envisioned as a secure login webservice for registered users. Registered users who can prove their identity as either authorized representatives of consortium members or commissioned services would be able access all the collected data, while other registered users would only be able to access fully open data. This is a consequence of the need that has arisen in the consortium for members to be able to act as data providers without providing their data completely without restrictions.

This means identity verification is a challenge that must be solved for external access. This may also go hand-in-hand with the development of external access to the data processing framework as a data analytics toolkit *service*. An intermediary step may also be to develop external read-only access to dashboards that can be created exploratory data analysis tools discussed in Sec. 4.6.1.

4.6 Data Processing

In the Triangulum context, the data processing framework should serve both the purpose of automating WP2 impact assessments and representing the WP5 internal deliverable Subtask 5.4.2. Consequently, both user friendliness and the analytical sophistication expressible by the data processing components must be addressed.

While the components described here have overlapping technology choices, it is important to clarify the distinction between the components on the level of designing the data processing framework in terms of the role each component is intended to play:

Exploratory data analysis (Sec. 4.6.1) is a pre-requisite for more advanced analytics, especially when the ontological data model of the data source is not well known, or part of the analyst's primary domain expertise.

Familiarity gained through exploratory data analysis provides the basis to build more advanced analytics. For example, a model can the be developed using a machine learning algorithm. Alternatively, a script can be developed to evaluate impact indicators for a given module, that simplifies the evaluation process from a human perspective by performing it in the background. Such processing is described as batch processing, in that it may be appropriate to perform operations on a larger subset of the data, and to undertake longer sequences of operations without manual intervention, as compared to exploratory data analysis.

Having developed a model — either by training a machine learning algorithm on a large dataset in batch processing or otherwise — it may be useful to turn this model into a modelling service that can take a smaller data set, such as the most recent history of a time series, or an individual data point from a given data source, and make a classification or prediction regarding the smaller data set or individual data point, based on the already developed model.

4.6.1 Exploratory Data Analysis Tools

The technology Kibana offers a flexible and user-friendly graphical user interface (GUI) as a web application that can be used to visualize and explore the data stored on an Elasticsearch cluster. In this case, the internal access is made using the APIs of Elasticsearch. Kibana is also part of the Elastic stack, and available under an Apache 2.0 license. However, unlike the other two Elastic stack technologies adopted in this design, Kibana does not require or lend itself to distribution over a cluster of multiple nodes. Thus, Kibana is chosen as a technology for exploratory data analysis in the UiS module implementation.

Regardless of its user-friendliness and usefulness for gaining a high-level overview of a data set, Kibana does not excel at sophisticated or computationally intensive data analysis. To address this requirement, Apache Spark (`spark.apache.org`) is chosen as a second technology for exploratory data analysis.

Spark accelerates Apache Hadoop and the MapReduce paradigm for distributed computing, by performing computations on data in memory. Spark is available under an Apache 2.0 license, and Arguably, Spark is most appropriate for a second level of exploratory data analysis, beyond simple visualization and going into formal statistical exploratory data analysis, as well as developing and testing individual operations that may later be combined into batch processing.

As a concession to user friendliness, Spark is here developed in terms of its Python API, called PySpark. Spark is also to set up JupyterHub (`jupyter.org`), which is available under a revised BSD license. JupyterHub can provide multiple users with Jupyter Notebook sessions, and can provide each user with a Spark session to work with dynamically provisioned Spark worker nodes. Note that Jupyter Notebook is a web application that enables writing and running code, visualizing results, and creating markdown annotations. The advantage over a typical integrated development environment is that the steps of a multi-step code can be more easily decoupled and evaluated individually.

Note that Spark must connect to Elasticsearch via a special Elasticsearch-Hadoop connector provided by Elastic.co, that enables bidirectional read/write capabilities for the Spark operations on the Elasticsearch cluster.

4.6.2 Batch Processing Tools

The purpose of batch processing has been explained above, and the technology to perform batch processing is chosen to be Apache Spark, as well. This takes advantage of the fact that exploratory data analysis can then more directly support the development of batch processing scripts. A distinction from the pure exploratory data analysis is that after the appropriate operations for batch processing have been developed, it makes sense to take the PySpark instructions out of the Jupyter Notebook and apply them as a simple Python file that is then submitted as a job to the Spark cluster via the command-line interface.

4.6.3 Modelling

As with batch processing, after developing the operations (and the model) using exploratory data analysis and/or batch processing, the PySpark code should be bundled a Python file that is then submitted as a job to the Spark cluster via the command-line interface.

However, to develop this into a service that is actively listening for input and ready to output on demand, it may make sense to manage such a script using Systemd, which is a technology included in the default operating system, Ubuntu.

4.7 Design Summary

With this description of the components design as derived from the system architecture in Chapter 3, a roadmap to implementing the actual software components is established. The technology choices for the necessary components inferred from the implementation units constitute the core of the system design here. The implementation of each component is less abstract, and adjustments are unavoidable. The specifics of how this design has been implemented are further described in Chapter 5.

Chapter 5

Implementation

This chapter presents the actual software implementation of the architecture and design discussed in the Chapters 3 and 4, respectively. The UiS module implementation is here discussed in terms of each of the implemented software components. This chapter provides further detail on the open source software technologies chosen for each component, and the configuration and coding undertaken to adjust and integrate the components. Where relevant, some context is also provided regarding the process of developing the UiS module implementation.

5.1 Pipeline and Components Overview

Before discussing the implementation of individual components in greater detail, an overview is in order to set a proper context for the implemented components. Note that some technologies (most of the "minor" technologies) are included in the implementation which are contingent on situational expediency and secondary to design choices (the "major" technologies).

It has been noted that the pipeline architecture pattern is a major part of the UiS module implementation, and the intended data flow can be explained here in greater detail to help motivate and give an overview to the subsequent sections focussing on the implementation of individual components. Thus, the pipeline and data flow can be summarized as follows:

1. The (external) data source offers data via some API or queue system that is specific to the data source in question.

2. The technical personnel at the data provider submit a "Data Intake Form" created in Google Docs so that the researchers at UiS can — among other things — develop an adaptor for the specific data source.

3. The adaptor is developed and instantiated as a running service on the cloud data platform.

4. The data is transferred on an automated, ongoing basis (regularly or irregularly) from the data provider to the cloud data platform.

5. The data coming into the cloud data platform is load-balanced by a Kafka cluster holding messages in queues.

6. The data in the Kafka queue is ingested and preprocessed by a Logstash cluster.

7. The data ingested and preprocessed by Logstash is indexed and stored in an Elasticsearch cluster.

   The data ingested and preprocessed by Logstash is also stored to S3/Ceph Object Storage for redundancy.

8. The data indexed and stored in an Elasticsearch cluster is accessible via the Elasticsearch API to data analysis tools.

9. The data in Elasticsearch can be accessed by Kibana, in particular for exploratory data analysis.

   The user can access the Kibana GUI via a browser navigating to the appropriate IP and port. Kibana can be used to build dashboards to visualize data.

10. The data in Elasticsearch can be accessed by Spark (using the Hadoop-Elasticsearch connector), in particular for batch data processing and analyses with advanced statistics and machine learning.

    The user can access a JupyterHub cluster to manipulate the data with Spark using the PySpark API. Spark can be used to calculate values of impact indicators based on data in the Elasticsearch cluster.

The major technologies adopted from external open source projects include: Kafka, Logstash, Elasticsearch, Ceph, Kibana, JupyterHub, and Spark. Minor technologies adopted from external open source projects that figured in the implementation include Filebeat (for system self-monitoring, log collection), Collectd (instance self-monitoring), EMQ (previously EMQTT, for Queue Adaptors), Systemd, and Zookeeper. In addition, various Python packages were used.

The minor technologies adopted from external open-source projects were used in specific components, and are discussed along with the major technologies in the corresponding subsections. This document assumes some technology as contextual and needing no introduction, such as the Linux operating system and the Bash shell and command language. In general, it is worth noting that Systemd is already built into the majority of the external open-source projects that are used as major technologies in the UiS module implementation. For example, when installed on Ubuntu, the Elastic stack technologies are managed as Systemd services. Also, while the Python packages used are discussed where appropriate, these are bordering on the aforementioned category of contextual technologies, where some familiarity must be assumed.

In this chapter, as well as in Chapter 6, specific files are discussed by referencing the path of the file with respect to the repository. For example, `cloud-data-platform/src/` contains the directories containing the deployment-specific implementations of the cloud data platform.

Table 5.1 shows how the key pre-existing technologies are involved with the major components in the module implementation. Parenthetical checkmarks in the table indicate

that while Systemd is a technology involved in the corresponding components, that involvement comes already implemented with the relevant technology.

	Data Acquisition	Queueing and Load Balancing	Data Ingestion	Data Storage	Data Access	Data Processing
Python	✓	–	–	–	–	✓
Systemd	✓	(✓)	(✓)	(✓)	(✓)	(✓)
EMQTT	✓	–	–	–	–	–
Kafka	–	✓	–	–	–	–
Logstash	–	–	✓	–	–	–
Elasticsearch	–	–	–	✓	✓	–
Ceph Object Storage	–	–	–	✓	✓	–
Kibana	–	–	–	–	–	✓
Spark	–	–	–	–	–	✓
JupyterHub	–	–	–	–	–	✓

Table 5.1: Overview of technologies in components

The rest of this chapter is divided into two divisions: the data collection framework and the data processing framework, in Sections 5.2 and 5.3, respectively.

In Sec. 5.2, the components of the cloud data platform that fall under the data collection framework, are described in detail. While not a deeply integrated component or even a software implementation, Sec. 5.2.1 discusses the aforementioned Data Intake Form used in Triangulum by UiS researchers. Section 5.2.2 discusses how the general Adaptor component is implemented with a custom python package that was created as part of the present work. In Sec. 5.2.3, the implementation of the queueing and load balancing component as a Kafka cluster is discussed. Section 5.2.4 goes through the implementation of the ingestion and preprocessing component as a Logstash cluster. Section 5.2.5 presents the implementation of the storage component as an Elasticsearch cluster and as S3/Ceph Object Storage. Concluding the data collection framework division of this chapter, Section 5.2.6 elaborates on the APIs of the storage component implementation, which provide the means of accessing the data collected.

In Sec. 5.3, the components of the cloud data platform that fall under the data processing framework, are described in detail. Section 5.2.6 discusses how the data processing framework accesses data stored by the data collection framework. Section 5.3 describes the exploratory data analysis tools implemented as both Kibana and clusters running JupyterHub and Spark.

5.2 Implementation division 1: Data collection framework

5.2.1 Data intake form

The exception to the rule about open source software was the use of the Google forms service to formulate the data intake form. Creating and hosting a survey service could in principle be done with open source solutions, but not in a timely manner alongside other priorities. The data intake form (`https://goo.gl/forms/72fRjJKgZ8fa1KHF3`) was used in Triangulum to collect the necessary information to generate adaptors for the various data sources — and to plan impact indicator calculations in terms of the data to be collected from those data sources.

The data intake form has been an important part of the iterative process of communicating with partners about what data they could provide, and under what conditions. The discussions surrounding this form also served as a starting point to determine the appropriate methods to establish ongoing data transfer from a given data source to the data collection framework.

The data intake form is also intended to collect human-readable / pseudocode level information that supports the manual creation of adaptors and impact indicators. This is necessary to connect a specific data source to the data collection framework. Also, information about impact indicators and the underlying data model is needed to program the automated calculation of those impact indicators for the given data source.

5.2.2 Adaptors

The software implementation of the cloud data platform, and especially the data collection framework, was undertaken concurrently with the process of discovering and characterizing the relevant external data sources. It was determined that all the data sources required individual *adaptors* to be developed in order to connect any specific data source to the general data collection framework.

To be able to create an arbitrary number of instances of the adaptor class of components adapted to transfer specific data sources, a custom Python package was created in a repository `cdpadaptor`, alongside the script `cloud-data-platform/generator/cdpadgenerator.py` which can be run to generate a new adaptor for a new data source, including a Systemd service (and timer where appropriate) for that adaptor.

The adaptor facilitates the data flow as follows:

The data is acquired and converted to a uniform encoding and serialization by the adaptors. The generated adaptors take the form of Python scripts that are executed by Systemd. Data points that are acquired by an adaptor are passed to the message broker implemented by Kafka, with a topic as given by the dataset name with the string `_raw` appended.

Logstash is a subscriber to various Kafka topics and will process the entries that are published to the various topics. All topics known to Logstash will do some processing to the entries and publish the processed entry back to Kafka under a new topic. Some topics are also configured to send the entries to storage, like Elasticsearch or S3.

As described in Chapters 3 and 4, the general data collection framework will require a dedicated adaptor component to each specific data source to account for its data model and appropriate data transfer method. Each external data source may have a unique API or message passing protocol, requiring individual adaptors. At the time of writing there are five external data sources for distinct Triangulum modules that are implemented with adaptors.

The primary behaviour of an adaptor is to download data from some external data source and convert that data to JSON format. This ensures that all data passed from the data acquisition subsystem enter the data ingestion subsystem (see Sec. 5.2.4) with a uniform encoding and serialization format. In general, an adaptor is implemented as a small Python script that is executed by one or more Systemd units. The adaptors either fetch data at predefined intervals using a Systemd timer, or implements its own responsive timing mechanism.

At an early stage of the project, the need for a variety of data transfer methods was anticipated. These data transfer methods, or options, were offered to the Triangulum partners:

- Option 1: Data provider hosts/provides a set of high availability brokers/queues to which cloud data platform subscribes.
- Option 2: Cloud data platform hosts/provides a set of high availability brokers/queues to which data provider may publish events.
- Option 3: Data provider posts data to RESTful API at cloud data platform.
- Option 4: RESTful API at data provider, cloud data platform performs frequent pull requests.
- Option 5: RESTful API at data provider, cloud data platform performs frequent poll requests.

However, implementation was done partly by prioritizing the available data sources. The custom Python package, `cdpadaptor`, contains superclasses intended to simplify the creation of new adaptors as new data sources become available. The subclasses in the `cdpadaptor` package were constructed according to the offered options above, as well as the initial experiences developing from the ground up adaptors for some available data sources. The classes of the package are described in further detail immediately below.

5.2.2.1 CDPAdaptor

The main class in the `cdpadaptor` package is named `CDPAdaptor`, and all adaptor classes inherit from it. The four adaptor subclasses in `CDPAdaptor` are:

- CDPPullingAdaptor
- CDPPollingAdaptor
- CDPQueueAdaptor

- CDPRESTAdaptor

These adaptor subclasses correspond to Option 4, Option 5, Options 1 and 2, and Option 3, respectively. Note that the subclass CDPQueueAdaptor can be used for both Option 1 and Option 2. In practice, Module 522 Energisentralen has been implemented using the CDPQueueAdaptor, together with an MQTT broker (implemented as EMQTT) hosted at the cloud data platform.

Each specific data source requires a specific adaptor, both to account for data source idiosyncrasies (with regards to API/protocol or data structure) and to assign appropriate topics to the data passed to the queueing and load balancing subsystem.

The `CDPAdaptor` class itself is never directly implemented as an adaptor for a specific data source. Instead, any adaptor for a specific data source must be implemented from one of the four subclasses. Which subclass is appropriate to implement an adaptor for a given data source should be determined from the subclass descriptions below.

The `cdpadaptor` package and its class inheritance are illustrated in Fig. 5.1. In the following discussion of the `cdpadaptor` package, refer to this figure as well as the corresponding code in the `cdpadaptor` repository.

Upon initialization, `CDPAdaptor` takes a list of optional key-value parameters through the argument `**kwargs`.

The optional parameter `kafka_topic` has a default value of `undefined_topic` and gives the Kafka topic to which the specified adaptor forwards its data. Note that while this parameter is technically optional, in practice it is mandatory. Note also that within Kafka, the value of `kafka_topic` defines a queue, although elsewhere in the cloud data platform, `kafka_topic` can be treated as an informative labelling of data according to its data source (module identifier and name) and degree of pre-storage processing (raw, sanitized, filtered).

A list of Kafka broker servers should be passed as an array of strings to the optional parameter `kafka_brokers`. However, this is only necessary if the default set of Kafka broker servers as defined in `/tmp/kafka_brokers.json` is to be overridden. If appropriate, the default set of Kafka broker servers may alternatively be overridden by passing as a string the path of an appropriate JSON file to the optional parameter `kafka_brokers_path`.

If `logging_disabled` is passed the value `True`, then log messages will be output to `stdout` rather than written to the specified log file. This useful for development and debugging of a data source specific adaptor.

The optional parameter `forward_method` is set to `kafka` by default, but can be overwritten with one of the following valid strings: Valid `forward_method` strings are `kafka`, which forwards to defined Kafka brokers, `cmd`, which forwards by printing to the command line with `stdout`, or `logging` which will add the data to the log file with log level `debug`. The alternatives `cmd` and `logging` are provided mainly for development and debugging purposes.

This makes it possible to develop an adaptor for a new data source outside the data center and without a Kafka instance by just using `forward_method="cmd"`. When the adaptor is ready it can reuse a well tested implementation of the `forward_method`.

The method `forward_data(self, data_str)` is implemented on the `CDPAdaptor`, and in most cases this implementation is to be used. This method will check the variable `self.forward_method`, and if it is set to `cmd` the `data_str` will be printed to the command

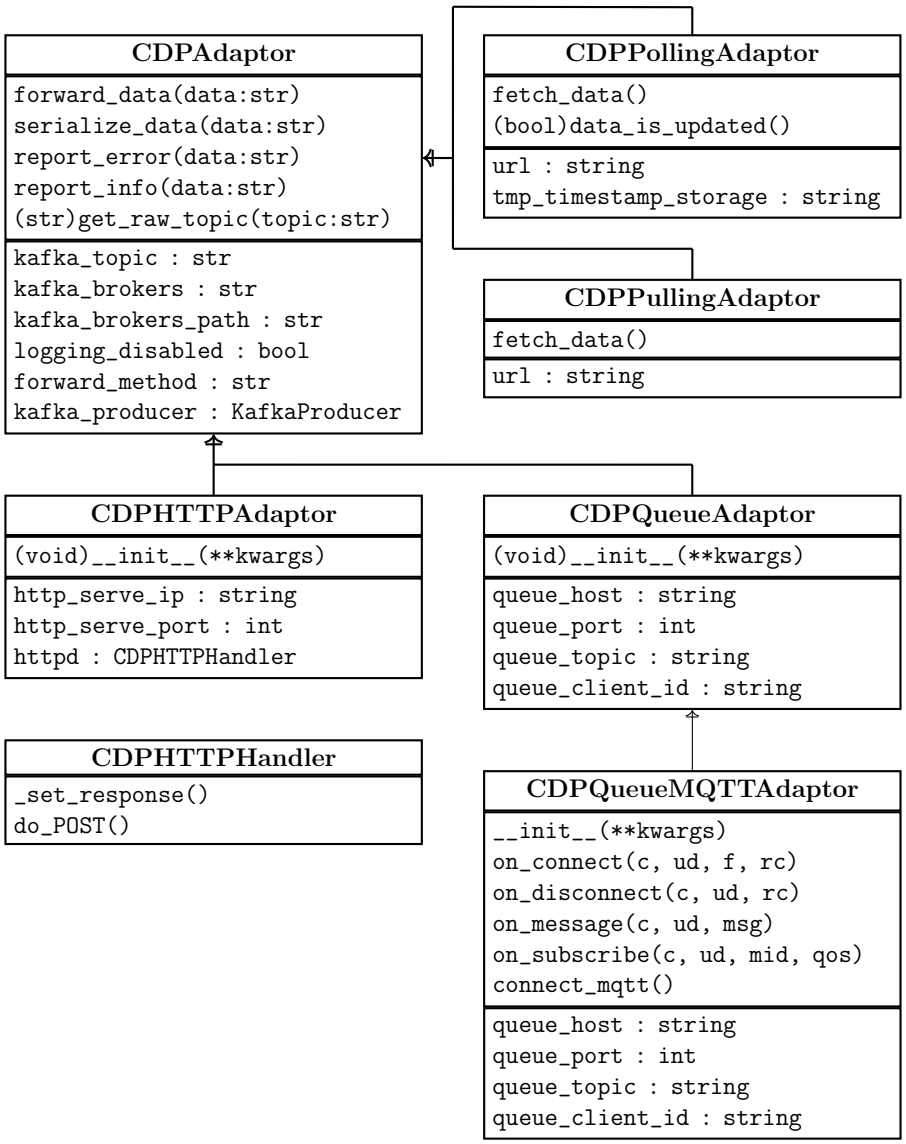

Figure 5.1: UML diagram for the `cdpadaptor` package.

line, if it is set to `logging` the `data_str` will be sent to the log file, and if it is set to `kafka` the `data_str` is sent to the Kafka broker servers.

The method `serialize_data(self, data)` is also implemented on `CDPAdaptor`. However, all it does is forward the data to `forward_data(self, data_str)`. This method is intended to be overwritten by subclasses if necessary, i.e. where the incoming data is not JSON format.

The static method `get_raw_topic(prefix):string` will take a string and append `_raw` to the end of it, to return a topic name consistent with the project naming convention. This will be further discussed in Sec. 5.2.3 and Sec. 5.2.4.

5.2.2.1.1 CDPPullingAdaptor

For data sources that serve data in a stateless manner, the simplest way to get the data is to *download any data that the source is serving* (to "pull") at regular intervals as appropriate for a particular data source. To accomplish this one can implement a subclass of the `CDPPullingAdaptor` class.

The `CDPPullingAdaptor` is a subclass of `CDPAdaptor` and implements one extra class variable, and one extra method. The class variable `url` can be passed upon initialization and indicates from where the data should be downloaded. The method `fetch_data(self):void` uses the Python package `requests` to make a `GET` request to the given url. For`fetch_data(self):void` other types of requests, this method must be overwritten. The response from the request is passed to the method `serialize_data`. If the data is in JSON format and no pre-processing is needed, the `serialize_data` method implemented on `CDPAdaptor` is appropriate. Otherwise, a different, appropriate the `serialize_data` method should be implemented.

5.2.2.1.2 CDPPollingAdaptor

The `CDPPollingAdaptor` is intended for data sources where the data should only be downloaded if it has been updated since the last time the data was downloaded. This is appropriate if a data source does not serve new data regularly or predictably. This can be handled either at the data source side or on the adaptor side. In some situations, the data source will provide some registration function and return an identifier that the adaptor will use with every request to keep track of what new data, if any, has been served.

The class `CDPPollingAdaptor` implements a class variable `url` that indicates where to get the data, and also a class variable `tmp_timestamp_storage` which is a path to the file that contains an identifying string (e.g. timestamp, or a hash) of the most recently downloaded data. This represents the adaptor keeping track of the data source updates.

The method `fetch_data(self):void` is almost identical to its namesake in `CDPPullingAdaptor`. However, in `CDPPollingAdaptor`, `fetch_data(self):void` checks the return value of the class method `data_is_updated(self):Boolean`.

If `data_is_updated(self):Boolean` returns `True`, the data will be downloaded, otherwise the program will exit.

`data_is_updated(self):Boolean` first checks if there is a file in the path `tmp_timestamp_storage`, and if so it parses the value in the file into a `datetime` variable `last_seen_dt`. Then a `HEAD` request is performed against the given `url` and the value of the header `Last-Modified` is compared against `last_seen_dt`. If the current content at the `url` is more recent than the last seen content, the file in the path `tmp_timestamp_storage` will be updated with the new timestamp and the function returns `true`.

5.2.2.1.3 CDPQueueAdaptor

In some cases the data is available by subscribing to a message passing queue system such as `Kafka` or `MQTT`. In this case, the `CDPQueueAdaptor` should be used to implement the data source specific adaptor. Two variants are possible: Either the adaptor subscribes to

an external queue broker server hosted by the data provider, or the cloud data platform hosts a queue broker server to which the data provider publishes data.

Either way, the adaptor must subscribe to a queue broker server, and for this the `CDPQueueAdaptor` subclass should be used to implement the adaptor. More specifically, if the queue system is a `MQTT` queue, the subclass `CDPQueueMQTTAdaptor` (a subclass of `CDPQueueAdaptor`) should be used.

`CDPQueueAdaptor` collects four extra parameters from the initialization. Again, these parameters are technically optional, but in practice mandatory to implement a data source specific adaptor. `queue_host` has a default value of `"localhost"` and refers to the host of the queue broker that the adaptor should subscribe to. `queue_port` gives the port number to use when connecting to the queue broker. The default value is `65534`. `queue_topic` is to be used as the topic to which the adaptor subscribes from the queue. `queue_client_id` is a string that in most cases can be used to identify the adaptor at the queue broker. The first three of the above four parameters must be set to match the corresponding values at the queue broker. The `queue_client_id` lets the adaptor identify itself to the queue broker.

As mentioned, in addition to the `CDPQueueAdaptor`, there is also a more protocol-specific adaptor class, `CDPQueueMQTTAdaptor`, that is made specifically to subscribe to a MQTT queue. Here we have a set of callback functions defined by the `paho-mqtt` Python package. This package enables MQTT communication.

The `paho-mqtt` callback methods `on_connect`, `on_disconnect`, and `on_subscribe` as implemented in `CDPQueueMQTTAdaptor` only report their respective event types to the log. The `paho-mqtt` callback method `on_message` as implemented in `CDPQueueMQTTAdaptor` is called when a message arrives on the topic to which the adaptor is subscribed. When this happens, the message is forwarded using the given data source specific adaptor's `forward_data` method.

The `CDPQueueMQTTAdaptor` method `connect_mqtt` creates the client and connects to the broker.

5.2.2.1.4 CDPHTTPAdaptor

The `CDPHTTPAdaptor` subclass enables the data provider to upload data to the cloud data platform through HTTP POST requests. Various conditions may justify the use of this subclass to implement and adaptor. For example, firewall restrictions at the data provider excluding the other subclasses. Also, if the data source is updated irregularly, this subclass represents an advantageous alternative to the `CDPPollingAdaptor` subclass. However, this subclass requires the data provider to actively send data to the cloud data platform. A dedicated port may be assigned to a data source specific adaptor implementing the `CDPHTTPAdaptor` subclass. The implementation of the data source specific adaptor should also specify the appropriate `kafka_topic`.

The initialization of `CDPHTTPAdaptor` collects the parameters `http_serve_ip` and `http_serve_port`, which gives the interface and port that the server will receive data on. The instance is then set to a global variable `parent`, meaning the instance is available everywhere in the current Python context. An instance of Python standard class `HTTPServer` is then instantiated and set to serve continuously. To handle responses, the `HTTPServer` uses the `CDPHTTPHandler` class defined in `CDPHTTPAdaptor.py`. Upon

a `POST` request, this will extract and decode the body of the request, and pass it to `forward_data`.

5.2.2.1.5 Executing the adaptors

All adaptors have a Systemd service that is responsible for executing the adaptor. Pulling and polling adaptors, also have a Systemd timer that handles the timing of when to execute the adaptor.

A Systemd service is defined by a configuration file, normally located at `/etc/systemd/system`. This file can have many properties, the most important ones for this use case will be presented here.

Listing 5.2.1: Contents of an adaptor systemd service file, `m432_vialistraffic.service.app`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

This is the service file for the data source `m432_vialistraffic`. The first block, `[Unit]` can have properties common for all units, in this case only the description is defined in this section. In the `[Service]` block the execution properties are defined. In this case, `ExecStart` gives the command to execute to start the service, and it will be run as the `User` and `Group` defined by the values of their respective keys.

This file service definition is very simple, this is because `m432_vialistraffic` is a pulling adaptor, and the service is thus started at intervals by a timer unit.

Listing 5.2.2: Contents of an adaptor Systemd timer file, `m432_vialistraffic.timer`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

This timer unit will start the service every 5 minutes, with an accuracy of 1 s. The accuracy can be set lower if needed. The service unit will start the adaptor, which in the case of pulling and polling will exit after one data download operation.

In the case of a queue or HTTP adaptor, the adaptor is listening for incoming connections from the data source. The service unit will then start the adaptor and make sure it is running. One configuration property that is added in this case is `Restart=Always`, enabling Systemd to restart the service if it for some reason dies.

For example, in the case of the adaptor for Module 522: Central energy plant (Subtask 5.2.2), and MQTT broker at the cloud data platform was required, and the data were published from the data source to that broker. Here, the timing of incoming data is controlled by the data source/data provider and may be intermittent.

Some data sources are continuously available for data transfer requests, but only update the available data periodically. In these cases, the timing of the data transfer is controlled by the adaptor. In practice, the useful interval for fetching data is constant for a given data source, based on the frequency of updates on the data source side, and the desired granularity on the cloud data platform side. Storage space is effectively not a limitation in the current implementation, and maximum time resolution without duplicates in the collected data is desired. For a continuous sensor type data source, some appropriate constant periodic sampling rate should be implemented.

Each adaptor has three functions: data transfer/fetching, re-serialization, and forwarding. Data transfer and re-serialization require customization to a specific data source to ensure compliance with the rest of the data collection framework, whereas forwarding is a standard sub-component that feeds the data forward into the next stage of the data collection framework.

Regarding re-serialization, note that different data sources are heterogenous in various aspects. One aspect of data source heterogeneity is serialization. Data may be serialized with codecs such as XML, JSON, CSV, or proprietary binary formats. To adapt the various data sources to the one data collection framework, all incoming data from external data sources is uniformly re-serialized to JSON if needed, which is amenable to ingestion by Logstash.

Each external data source may have a unique API or message passing protocol, which in turn requires the development of an individual adaptor to collect data from that data source.

All adaptors inherited from the Adaptor class are implemented with re-serializing and data forwarding methods as shown in Fig. 5.1. In addition, each adaptor will have the properties of topic name.

Each data-source-specific adaptor inherits from an Adaptor subclass that reflects the data transfer options selected by the respective data providers in the data intake form. For example, KolumbusVMAdaptor inherits from the PollingAdaptor subclass of the Adaptor class. The data-source-specific adaptor instantiates itself.

5.2.3 Queueing and load balancing

Section 4.2.2 discusses how queueing and load balancing is a required function in the UiS module implementation, and why Kafka was selected as the technology basis for this component. The role of this component is to buffer the data that is captured with various adaptors from their respective data sources, and to dispatch the data to the data ingestion component.

The data stored in a Kafka cluster is distributed over partitions, which are distributed over the nodes (servers) in the cluster. Each node usually holds several partitions. Each Kafka node can handle the data operations and requests for the partitions it holds. The data belonging to one specific topic in Kafka can be divided among several partitions, and thus several nodes. All partitions in the cluster are replicated across several nodes to make the system fault tolerant. Each partition in the cluster has one node elected as leader, and the leader will handle all read and write operations for that partition. Should the leader go down, another node will be elected leader. Leader election and communication

between Kafka nodes is handled by Zookeeper.

These nodes run on separate computers or VMs. Thus, they have different IP addresses and accept communication through a specific port, where the default port is `9092`. A Kafka node decides on the message size limit (configurable), which can be used to avoid performance bottlenecks when messages are too large. A Kafka node also decides on the period of time a message should be kept (also configurable), which can be used to provide a recovery buffer, since as long as they are kept, the messages can eventually be forwarded to the consumer. In the UiS module implementation, this mean the data ingestion component, implemented with Logstash.

Furthermore, a Kafka node decides on the *replication factor* of the messages it receives. This means how many copies of the received messages should be held on other nodes. This is also how Apache Kafka supports fault tolerance. To guarantee a certain replication factor, a Kafka node must be able to communicate with other Kafka nodes and hence Zookeeper.

This is managed by the deployment strategy in Ch. 6, and simply consists of providing a list of Zookeeper nodes and their IPs to each Kafka node. Because a Zookeeper node can also fail, Zookeepers should also be deployed as a cluster. Put together, a minimum set a Kafka nodes to achieve a replication factor of 2 when one Kafka node fails has to be at least 3 Kafka nodes. Likewise, to be able to determine a partition leader, a Zookeeper quorum must be reached. To tolerate the failure of a single Zookeeper node, the Zookeeper cluster needs at least 3 nodes to reach a quorum.

All the data acquired by the adaptors is forwarded to one of several Kafka brokers, selected in a round-robin fashion. Kafka is then in charge of dispatching the data to the Logstash nodes for preprocessing and ingestion downstream towards storage in Elasticsearch and S3/Ceph Object Storage. The adaptors publish data as messages to Kafka in topics named according to the specific data source. On the other side, the Logstash nodes are subscribed to specific topics among those that have dedicated adaptors implemented and running as a service.

5.2.3.1 Parameters affecting fault tolerance

The queue and load balancing component has Kafka parameter values that affect fault tolerance. These parameters are shown in Listing 5.2.3.

Listing 5.2.3: Contents of Kafka server configuration file, `cloud-data-platform/src/dep-openstack/salt/configs/kafka/server1.properties`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

Note that all the Kafka nodes have the same parameter settings as in the `server1.properties` file shown in Listing 5.2.3, except for those parameter relating to that node identifying itself to other nodes in the cluster. For example, `broker.id` and `advertised.listeners` are specific to the Kafka node with those values, but all the other parameter values should be the same across all the Kafka nodes in a given cluster.

Note also that the cluster-specific sets of IPs reflect a specific intended deployment with a specific set of nodes in each component cluster/functional group.

Each topic consists of a stream of messages or records. These messages are stored across one or several partitions which each maintains a sequence of messages. Each unique message is stored in only one partition. However, the partition is replicated across several Kafka nodes. Only one Kafka node is the leader of the partition and manages read/write requests, while the other Kafka nodes replicate that partition passively to ensure fault-tolerance and obey read/write requests passed on via the leader.

Thus, the *number of partitions* and *replication factor* are two parameters affecting the component fault tolerance.

The `zookeeper.connect` parameter in the above listing is crucial for the *managed leader election* of the Kafka cluster. Each Kafka node must be informed of the Zookeeper cluster that manages leader election for the Kafka cluster. In the above listing, `zk1`, `zk2`, and `zk3` contain the IPs of the nodes in the Zookeeper cluster.

The log retention period (`log.retention.hours`) sets how long messages in a partition is kept. This parameter is needed by Kafka because it is stateless, meaning the consumers must maintain a state regarding what messages have been consumed from the Kafka queue. Kafka simply deletes messages that have been held in the queue for a duration equal to or greater than the log retention period.

The benefit of this stateless queue is that despite failures among Kafka nodes or consumer nodes, the available history of messages is always complete from the present moment and back to one log retention period ago. Thus, this parameter affects the component attribute recoverability (resiliency or ability to recover from failure) for the consumer of the queue. In the module implementation, this means supporting the Logstash component responsible for the function of ingestion and preprocessing.

As for *message size* (or "maximum size of a request"), this is limited by the setting `socket.request.max.bytes`, which supports the attribute *resilience* by protecting against out-of-memory failure on the Kafka node. This means `socket.send.buffer.bytes` and `socket.receive.buffer.bytes` decide the size of the packets into which a message is divided when Kafka nodes send and receive these messages, respectively.

5.2.3.2 Logstash parameters relating to Kafka

As illustrated in Listing 5.2.4, subscribing nodes are identified with a `group_id` which is subscribed to a particular (set of) topic(s).

When a new message is to be published on a given topic, only one of the nodes in a subscriber group will be sent a copy of that message. This is a consumer feature. Thus the consumer must determine its own load balancing strategy.

Listing 5.2.4: Selected contents of Logstash pipeline configuration file, `cloud-data-platform/src/dep-openstack/salt/configs/logstash/conf.d/coinbase.conf`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
```

```
$ sudo -H pip3 install .
```

5.2.3.3 Load balancing the publishing adaptors

Recall that a `CDPAdaptor` uses the package `cdpadaptor` to forward data to Apache Kafka nodes. This is done by calling the method `forward_data(data:str)` as shown in the class diagram from Figure 5.1.

The Kafka server list in `/tmp/kafka_brokers.json` is part of the cloud data platform and is managed by the deployment strategy as explained in Ch. 6. This list of nodes and IPs defines the queuing and load balancing component to which `CDPAdaptors` will load-balance *publish/send* operations in a round-robin manner. This is provided by the official `kafka` package for Python.

5.2.3.4 Load balancing the subscribing Logstash nodes

This section discusses how the load balancing component consisting of the Kafka cluster interact with the ingestion component consisting of the Logstash nodes.

Listing 5.2.5: Selected contents of Logstash pipeline configuration file - subscriber group, `cloud-data-platform/src/dep-openstack/salt/configs/logstash/conf.d/coinbase.conf`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

In Listing 5.2.5, we show one section of an example Logstash pipeline configuration file (`cloud-data-platform/src/dep-openstack/salt/configs/logstash/conf.d/coinbase.conf`) where `kafka1`, `kafka2`, `kafka3` and represent the IPs of the Kafka nodes in the cluster.

5.2.4 Data Ingestion

After acquiring data from external data sources and placing these in the queueing and load balancing component, the data ingestion component can consume the data points as messages from Kafka topics that Logstash nodes subscribe to, in order to ingest the data into storage. For each data source, a dedicated Logstash pipeline configuration file is defined under `cloud-data-platform/src/.../configs/logstash/conf.d`.

Logstash prepares the data for storage. All incoming data exist in three stages of preparation: raw, sanitized, and filtered. The data arrives as raw data, and is then sanitized before it is filtered. Data can be obtained from the Kafka queue at any of these three stages.

5.2.4.1 Sanitizing

The first step of data preparation is here called sanitizing and it will convert data that is in state `raw` to state `san`. Logstash will here subscribe to the Kafka topic for a data source that ends with `_raw`, sanitize the data entry before publishing it back to the Kafka queue with the topic ending in `_san`.

This processing step is intended to do the operations needed for all possible uses of the data. For example, anonymization may in theory be feasible at this stage. This processing step is supposed to be done before storing any data. The data that has topic ending with `_san` is intended to be the closest to the raw data that can be legally or safely used in the system.

Logstash will store all the data ending with `_san` to S3/Ceph Object Storage and Elasticsearch.

5.2.4.2 Filtering

The filtering step of the data preparation is meant to do more intrusive and application specific data transformation, for example converting values and types belong here. Since the data was stored in the last step, it is safe to alter the data in this step, since it then is reproducible.

Data entries with topic ending in `_filtered` will be stored to Elasticsearch.

5.2.5 Data Storage

5.2.5.1 Indexing

The primary data storage component is implemented using Elasticsearch, which enables the indexing of data points — passed as messages from Logstash — into searchable document collections. Elasticsearch is also distributed and each node must be configured with the list of nodes that constitute its cluster, which is set in the `/etc/elasticsearch/elasticsearch.yml` file on the node.

Listing 5.2.6: Uncommented contents of Elasticsearch configuration file, `cloud-data-platform/src/dep-openstack/salt/configs/elasticsearch/elasticsearch.yml`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

Listing 5.2.6 shows an example Elasticsearch configuration file. The configuration of the cluster can be made very simple, although settings of the Elasticsearch installation with respect to each host node are also crucial. These include dependencies on correct Java Developer Kit versions and appropriate memory allocations on the host node.

5.2.5.2 Object Storage

S3/Ceph Object Storage is provided directly by CCP. Data is passed to Ceph Object Storage primarily for backup and archival purposes.

5.2.5.3 Monitoring

For a running system, there are a large number of events that can cause it to stop running, or perform lower than expected. To keep track of the system's general performance, and wether it is running or not it can be nice to monitor the various aspects. Self monitoring of the system is done by collecting, analyzing and visualizing three types of data:

- Application logs
- Application metrics
- System metrics

5.2.5.3.1 Application logs

All applications that run in the system produce log files that are stored locally at each virtual machine. By using a small application called Filebeat from the Elastic stack, we are able to watch these log files and send every new entry immediately to Logstash. Logstash is then configured to parse the text formatted log entries by using *grok* filters. This will convert the log entries to structured data so that each field can be used for searching later. The parsed log events can then be indexed in Elasticsearch. This makes debugging easier than if one had to ssh into each machine to look at log files. It also makes it easy to set alarms that can alert developers when something unexpected happens. For example, if an error message is produced at a node, developers can get a chat message and be able to respond to the event much faster than previously possible. These logs can also be used to generate visualizations that show how the system is operating, and the number of messages for the various error levels can be a good measurement of how well the system is operating.

5.2.5.3.2 Application metrics

Some of the applications in the system also produce application metrics. These are measurements of application performance and similar that are reported by the application in real time. Collecting these metrics make it possible to visualize the actual, real time performance of each application in the system. Sudden changes in the various metrics can in some cases be as sign of something not behaving normally and thus cause an alert to developers. This can leverage fixing problems before they turn into real problems and error entries in the applications log.

5.2.5.3.3 System metrics

Every virtual machine in the system is also monitored by a small application called Collectd, short for *collect daemon*. This application will collect system values at predefined

intervals and report them to Logstash which in turn will forward the data to Elasticsearch. This kind of metrics can give valuable information like how much free memory and disk space each VM has, how much CPU any process is using, or how much traffic is going in and out of each of the network interfaces. This is also a kind of preventive monitoring, in that it makes it possible to catch and fix problems before they become dangerous, critical, or harmful. For example a disk that is filling up might be forgotten if it was not for Collectd's constant monitoring of the free space on all disks. Applications that seem slow and unresponsive might be running on a VM that has little or no available memory.

5.2.5.3.4 Visualizing

Visualizations of metrics are created to give developers and other users a quick way to identify the system status and performance.

5.2.6 Data Access

The Elasticsearch API is a RESTful API that provides access to the collected data, as well as cluster status information. The API can be accessed with CLI, scripts (e.g. Python scripts), Kibana, and the Elasticsearch-Hadoop connector, to name the most relevant methods for the UiS module implementation.

While S3/Ceph Object Storage is operational and in use by data ingestion processes, the development of processes accessing the data subsequently from this secondary data storage component has not been a priority so far.

5.3 Implementation division 2: Data processing framework

5.3.1 Exploratory Data Analysis Tools

The most user-friendly component of the exploratory data analysis tools is the Kibana GUI. Kibana accesses the Elasticsearch cluster determined in the configuration file `/etc/kibana/kibana.yml` on the node where Kibana installed.

Listing 5.3.1: Uncommented contents of Kibana configuration file, `cloud-data-platform/src/dep-openstack/salt/configs/kibana/kibana.yml`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

Listing 5.3.1 shows the Kibana configuration file, which only needs to specify a single Elasticsearch node through which Kibana can access the data on the whole Elasticsearch cluster. Note that the Kibana configuration also can specify the tile map that can be used to visualize geography related to GPS position data.

The more compute-intensive implementation of the data processing framework, is implemented as co-located JupyterHub and Spark clusters. The JupyterHub web application provides multi-user sessions of Jupyter Notebook. This capability arguably makes the Spark computing system user-friendly enough for exploratory data analysis. The JupyterHub/Spark cluster is then enabled to provide PySpark sessions with as shown in Listing 5.3.2.

Listing 5.3.2: Enabling PySpark, `cloud-data-platform/src/jupyspark/salt/configs/pyspark/kernel.json`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

Note the need to declare a Spark cluster master, and for Spark worker nodes to be configured with IP (`10.0.0.17` in this example) and port (`7077` by default) of the Spark master node. However, the Spark master node does not need to be configured with information about Spark worker nodes. Thus the Spark worker nodes can be created and join the Spark cluster without interfering with the Spark master configuration.

The Hadoop-Elasticsearch connector file, `elasticsearch-hadoop-5.5.1.jar` file is downloaded and made available to each of the Spark worker nodes with a local copy in the directory `/opt/sparkjars/`, enabling the loading of Elasticsearch API query results into the Spark data structure Resilient Distributed Datasets.

Another aspect of Spark that enhances its applicability to exploratory data analysis is that it is also supported with a library (Spark SQL) for working with data in a similar way to Python library `pandas`, with DataFrames on top of the Resilient Distributed Dataset structure.

5.3.2 Batch Processing and Modelling Tools

The use of Spark for the components of exploratory data analysis, batch processing, and modelling are distinguished in Sec. 4.6.1, Sec. 4.6.2, and Sec. 4.6.3, respectively. However, in contrast with JupyterHub makes Spark user-friendly enough for exploratory data analysis, the CLI command
`$ /bin/spark-submit script.py` can be used to run a prepared set of PySpark commands on the Spark cluster. This is the most appropriate approach to both batch processing and modelling applications after initial development.

Spark is also supported with libraries for machine learning (MLlib) and processing real-time streaming data (Spark Streaming), which may be especially appropriate for Batch Processing and Modelling applications, respectively. However, the Spark library for processing data with a graph structure, GraphX, is not well supported for the PySpark API compared to the Java and Scala APIs.

5.4 Implementation Summary

This chapter has presented selected details of the implementation of the components implemented in the present work. In general, the technologies chosen must be configured tointeractappropriately. Thecomponentsimplementedasdescribedinthischaptercan be deployed in various ways, which is further elaborated in Chapter 6.

Chapter 6

Deployment

6.1 Overview

Given a computing problem, a software solution can be developed through the steps of architecture, design, and implementation. With such software implementation completed, the problem can only be satisfied in practice if the software is installed and run on some computing hardware. This constitutes deployment.

The software implementation may be installed and configured manually, taking into account the pre-existing conditions, such as: The computing hardware, its operating system, other software, and the communication network. However, this is a time-consuming, error-prone, and laborious task.

In reality, a modern development process involves repeated iterative cycles of development, deployment, and testing. In recent years, this has lead to a tighter coupling of software development and operations activities, which combine into a practice called DevOps.

Thus, if the deployment process can be automated, time can be spared and errors can be solved once rather than repeatedly.

One condition for automation is that the hardware and software context to which an implementation will be deployed get standardized. Consistent choice of virtual machine configuration and operating system satisfies this condition.

More specifically, deployment may involve the processes that download, install or upgrade, configure, and set up persistent services based on the components in the implementation.

For the systems presented here, deployment occurs in the context of installing the software implementation on a pool of distributed computing hardware resources. The interface between the deployment and the hardware resource pool may be a cloud computing platform. In the present work, this cloud computing platform (CCP) was based on OpenStack.

In this chapter, as well as in Chapter 5, specific files are discussed by referencing the path of the file with respect to the repository. For example, `cloud-data-platform/src/` contains the directories containing the deployment-specific implementations of the cloud data platform.

This chapter details the automation of processes that perform deployment of the cloud

data platform and data analytics toolkit to the cloud computing platform. Sec. 6.2 further details the approach to the deployment work and the pre-existing technologies used. From the outset, both the cloud data platform and data analytics toolkit modules have been architected for a distributed computing (i.e. multiple-node) implementation. At the same time, the development process naturally started with simple working systems, and increased in complexity as the required properties depending on distributed machines were implemented. The subsequent sections in this chapter detail the corresponding stages of increasing complexity in deploying the system. Sections 6.3 to 6.4 go into detail about the progressively more complex deployment schemes. Finally, Sec. 6.5 addresses how alternative deployments may be customized as needed.

6.2 Methodology and technology

6.2.1 Methodology

The approach to deployment in the present work is based on the following assumptions: Relative to the deployment, the implementation exists as a static code repository. In other words, this project does not undertake continuous deployment. However, the implementation is deployed to some computing resources – hardware or virtual machines – to instantiate the implementation as an active system. This means that the implementation may be deployed to an indefinite number of computing resources, and is thus capable of producing an arbitrary number of instantiations.

The present work configures deployment for automatic execution, thus enabling an arbitrary horizontal scaling of deployments with constant work. A good deployment system is one that is general enough to be easily reproducible, yet customizable. The methodology to achieve this consists of configuring deployment as machine-readable recipes for required conditions and software. These machine-readable recipes can be distributed to the nodes. The machine-readable recipes for deployment must be written in a syntax readable by some appropriate interpreting software. This interpreting software must itself be deployed to the computing resources first. In addition, the interpreting software may work as a distributed system, and may require the assignment of some master node(s) to coordinate the deployment across the multiple targets. The multiple target nodes of the deployment may consist of hardware machines or virtual machines.

6.2.2 Technologies

The machine-readable recipes for deployment must be readable by some appropriate interpreting and deploying software. The present work has relied on SaltStack software to mediate the various system deployments. Thus, in general the machine-readable deployment recipes have been written as `.sls` files, which are native to SaltStack. Possible alternatives to SaltStack include Ansible, Chef, and Puppet. SaltStack uses a declarative approach and defines a state which the machines should enter, as opposed to an imperative form of recipes used by e.g. Puppet or Bash scripts. SaltStack itself is deployed to the relevant nodes using a Bash script, and the nodes are defined as Salt master and Salt

minion nodes. The Salt master node(s) accept Salt minions as part of their cluster, and establishes SSH sessions from the master to each of the minions.

These encrypted channels allow the Salt minion nodes to trust directives from the Salt master node(s). Subsequently, the master nodes direct the minion nodes to enter a state described by some machine-readable deployment recipes (`.sls` files). These states may be described by a combination of `.sls` files, and different states may describe different functionalities that the various minion nodes should assume. In the simplest case, a node may act as both Salt master and Salt minion, and deploy to itself the implementation of cloud data platform and data analytics toolkit. This is further described in Sec. 6.3.1 and Sec. 6.4.1. However, multi-node clusters can also be deployed to develop and take advantage of the fault-tolerant, distributed paradigm of the technologies in the module implementations. This is further described in Sec. 6.3.2 and Sec. 6.4.2.

Vagrant is intended to support software development by facilitating rapid and portable replication of a software development environment. In combination with SaltStack, Vagrant enables simplified, programmatic provisioning of virtual environments on a computer. For local physical machine hosts, the present work has used Vagrant in combination with VirtualBox as a hypervisor. Both Vagrant and VirtualBox are open source, available under the MIT license. Alternative hypervisors (not necessarily open source) that can be used with Vagrant include Docker (remove Docker), HyperV, and VMWare. The combination of Vagrant and VirtualBox represents a convenient context to develop and test deployment recipes using SaltStack. Virtual private networks are enabled and support clusters of multiple virtual machines. However, this is not expected to support a stable production environment.

For a stable production environment, the module implementation is deployed to virtual machines provisioned with OpenStack. OpenStack is a cloud computing platforming toolkit that can be used to manage cloud computing resources. In the Triangulum project, the cloud data platform and data analytics toolkit modules are instantiated on the UiS cloud computing platform (CIPSI computing platform, CCP), which uses, among other technologies, OpenStack to manage the data centre hardware computing and network resources. The hypervisor used with OpenStack on CCP is called KVM (Kernel-based Virtual Machine).

6.3 Local virtual machine environment with Vagrant and Virtualbox

6.3.1 Deployment to single VM locally

This subsection will discuss the simplest case of deployment using SaltStack. Here, the module implementation is deployed to a single Vagrant virtual machine set up as its own Salt master and minion.

This deployment scheme was developed concurrently with the overall implementation, especially the data collection framework (see Sec. 5.2). This enabled the initial development of the implementation to happen with minimal attention to networks and fault-tolerance, while at the same time using fault-tolerant technologies. This helped to establish a baseline

functional system that is also intrinsically scalable to a multiple node deployment.

The prose descriptions in this subsection will use file paths relative to the repository directory `cloud-data-platform/src/`, since the contents of both `cloud-data-platform/src/dep` and `cloud-data-platform/src/vagrant` are used. For example, this subsection may discuss the files `cloud-data-platform/src/dep/salt/top.sls` and `vagrant/Vagrantfile`, and in all prose descriptions, these paths will be given relative to `cloud-data-platform/src/`. This is done both for brevity and clarity.

The case of using Vagrant and VirtualBox on a local computer to provision a virtual environment with a single node is illustrated in Fig. 6.1. The cloud data platform and data analytics toolkit can be deployed to the single, locally hosted virtual machine.

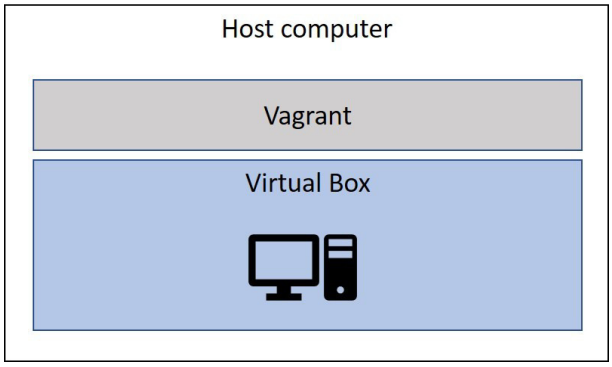

Figure 6.1: Provisioning single VM node locally.

6.3.1.1 Key files

In the file `cloud-data-platform/src/vagrant/Vagrantfile`, shown in Listing 6.3.1, the code (omitting comments) provisions a single virtual machine with Vagrant and VirtualBox, and deploys the cloud data platform to it using SaltStack.

Listing 6.3.1: Configuration for provisioning a single VM node locally, `cloud-data-platform/src/vagrant/Vagrantfile`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

The above file represents a recipe to create a virtual machine which we call "masterless" within the Vagrantfile, but once instantiated the VM will have the hostname `saltmaster.local`. The VM is to be provisioned with 2 virtual CPUs, 4 GB memory, and Ubuntu 16.04 operating system. The virtual machine synchronizes with two folders on the host machine: `cloud-data-platform/src/dep` to get the cloud data platform implementation to be deployed on a single node, and `../../vstorage` which can be used to store raw data as backup files. A private virtual network is defined to forward ports between the external and internal networks. The Kibana web interface is made accessible through port 8080, and Elasticsearch through port 9201.

Next, the SaltStack deployment is configured for the masterless VM. SaltStack is used to deploy the implementation to the VM upon instantiation, but SaltStack will not subsequently be available on the masterless VM. Note that the Vagrantfile provisions a VM which is "masterless", meaning the same node is both the only master and only minion with respect to this deployment.

The Vagrantfile refers to a minion configuration file shown in Listing 6.3.2 with three uncommented lines:

Listing 6.3.2: Configuration for single VM node as minion, `cloud-data-platform/src/vagrant/etc/minion`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

The Vagrantfile also refers to a master configuration file shown in shown in Listing 6.3.3 with two uncommented lines:

Listing 6.3.3: Configuration for single VM node as master, `cloud-data-platform/src/vagrant/etc/master`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

These latter two files represent a near-minimal configuration to support the being its own master and slave. Informally, this can therefore be called a "masterless" In other deployments, the minion and master files of VMs participating in a multi-node cluster would require more details to organize the relationships that enable deployment using SaltStack.

In the above Vagrantfile, the line `salt.run_highstate = true` applies the state described in the file `top.sls` (shown in Listing 6.3.4) and its dependencies to the masterless VM.

Listing 6.3.4: Salt state for local single VM node deployment, `cloud-data-platform/src/dep/salt/top.sls`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

Each of the lines indented under `'*'` refer to another `.sls` file in the same folder. These are called in the order they appear above, when the `'*'` state is applied to a machine. This section will briefly go through some of these files called by `top.sls`, starting with `common.sls`, shown in Listing 6.3.5.

Listing 6.3.5: "Common" state for local single VM node deployment, `cloud-data-platform/src/dep/salt/common.sls`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

The `common.sls` file is the first called in `top.sls` because it ensures the installation of packages on the machine that are necessary for subsequent `.sls` files to be correctly applied. The packages include `htop` and `nload` serve monitoring functions, which is helpful for troubleshooting. The package `rubygems` was required to install the logstah-jmx plugin for monitoring Apache Kafka. Java Developer Kit (`openjdk-8-jre`) and `apt-transport-https` are included as a pre-requisite for some Elastic stack installations.

This provides some insight into the structure and function of `.sls`-files. For each of the following lines in `top.sls`, a corresponding `.sls` file in the same folder is called and applied to the machine. Based on these files, packages are installed, folders and files are managed, and Systemd services are set to run. We will address the these files with only a brief description of each.

`kafka.sls` sets up the load-balancing component of the cloud data platform as a running Apache Kafka instance. This is in turn managed by a Zookeeper Systemd service.

`elasticsearch.sls` installs Elasticsearch and copies a pre-configured YAML file from the repository. Then a Bash script, `createindices.sh` is used to index expected data sources. These have files with corresponding Elasticsearch mappings stored in the `cloud-data-platform/src/dep/salt/createindex` folder.

`logstashprepare.sls` sets up folders and imports configuration files that are needed for the ingestion component of the cloud data platform, Logstash. This preparation is performed before Logstash is actually installed, and even before the general data acquisition state `acq.sls`, `cdpadaptor.sls`, and the data source specific states `mXYZ-***.sls` are applied. `logstashprepare.sls` sets the stage for the subsequent components by inserting various configuration files, directories, and a script used by adaptors to optionally store data to the S3 object storage. This salt state file begins the sequence of preparations leading up to `logstash.sls`. These preparations include deploying the data source specific adaptors which will be connected to the general data collection framework.

`acq.sls` ensures that the necessary standard Python versions and packages are installed. Also, the `acq.sls` file places into the target VM a `PollingClient.py` script that implements a parent class for adaptors that poll for data from some source, fetch the data, and forward the data to either Logstash or Kafka. This legacy implementation precedes the `cdpadaptor` implementation of data source specific adaoptors. `PollingClient.py` is used in the current `m531_kolumbus_vm` adaptor, although this approach is deprecated in favour of `cdpadaptor` and the `m531_kolumbus_vm` adaptor will be updated in future work.

`cdpadaptor.sls` installs the custom Python package from its own repository. The `cdpadaptor` package was developed in this project, and is discussed in Sec. 5.2.2.1.

The data source specific adaptors `mXYZ-***` are each represented by an `.sls` file, that in turn each applies a pre-configured Logstash pipeline configuration file, a (`cdpadaptor`)

implementation of the specific adaptor as a Python script, a systemd service, and a systemd timer. Note that a timer is not included in the case of `CDPQueueAdaptor` implementations, such as `m522_energisentralen`, nor in the case of `CDPHTTPAdaptor` implementations.

`logstash.sls` installs Logstash with a pre-configured YAML configuration file, including the JMX input plugin, which enables the collection of remote Java applications' metrics. In the cloud data platform, the JMX input plugin is used to monitor Kafka.

`kibana.sls` and `filebeat.sls` install Kibana and Filebeat, respectively, each with a pre-configured YAML configuration file, and make these run as systemd services.

Finally, `collect.sls` installs and sets to run as a service the collectd logging of machine self-monitoring. Note that the configuration files of Filebeat and collectd are not set up for Vagrant deployment. In the Vagrant single-node deployment discussed in this section, the collectd service will appear silent to the Filebeat service listening for self-monitoring data to forward to Elasticsearch.

6.3.1.2 Operations to enact the deployment

In the system where Vagrant is installed, the single node deployment can be done by going to the location of the above `Vagrantfile` and entering the commands shown in Listing 6.3.6 (after the chevron `$`) in the command prompt.

Listing 6.3.6: Commands to enact deployment of single VM node deployment in a local environment.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

Vagrantfile takes care of the deployment described by the key files.

6.3.1.3 Notes on context

For further details, refer to the specific files in the repository, the appropriate sections in Ch. 5, and the official documentation of the corresponding software.

Note that the same SaltStack files, i.e. the contents of `cloud-data-platform/src/dep/salt` can be used to deploy the same implementation to any single physical or virtual machine with a compatible operating system (default: Ubuntu Xenial 16.04) and SaltStack installed.

6.3.2 Deployment to multiple VMs locally

This subsection will discuss a slightly less simple case of deployment using SaltStack. Here, the module implementation is deployed to multipole Vagrant virtual machines set up with one Salt master and several Salt minions. Both as a Salt cluster and as an active deployment of the implementation with articulated functional groups, this deployment case is most notably different from the single node deployment in the previous by virtue

of the requirement to coordinate the communication between the nodes in the cluster to exploit the distributed computing paradigm of the various technologies.

The case of using Vagrant and VirtualBox on a local computer to provision a virtual environment with multiple nodes is illustrated in Fig. 6.1.

The prose descriptions in this subsection will use file paths relative to the repository directory `cloud-data-platform/src/dep-openstack`, since the contents this directory specifies both the necessary provisioning with Vagrant and the deployment using SaltStack. For example, this subsection may discuss the files `salt/top.sls` and `Vagrantfile`, and in all prose descriptions, these paths will be given relative to `cloud-data-platform/src/dep-openstack`. This is done both for brevity and clarity.

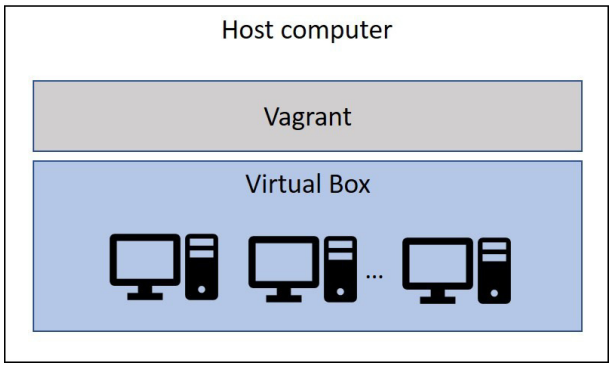

Figure 6.2: Provisioning multiple VM nodes locally.

Sec. 6.3.1 presented how to deploy CPD on a single machine using vagrant.

For a multiple node deployment, Vagrant can automate the provision of a set of virtual machines and their private network. Note that Vagrant should primarily be used for testing in the course of development, and not in production. In the present case, we have limited the deployment to 16 VMs. These VMs must all have salt-minion installed and should be known to a salt-master node.

6.3.2.1 Key files

In the `Vagrantfile` for this deployment, shown in Listing 6.3.7, the 16 VMs are provisioned with Vagrant and VirtualBox, and deploys the multi-node version of the cloud data platform to these VMs using SaltStack. The Vagrantfile reads the file `cloud-data-platform/src/dep-openstack/servers.yml`, which lists VM names and their respective IP addresses as provisioned in a private network by Vagrant. This creates a SaltStack cluster.

Applying the distributed set of Salt recipes `cloud-data-platform/src/dep-openstack/salt/*.sls` from the master node will deploy a distributed implementation to the virtual cluster.

Listing 6.3.7: Configuration for local multiple-VM-node deployment, `cloud-data-platform/src/dep-openstack/salt/Vagrantfile`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

As mentioned, the above `Vagrantfile` calls `servers.yml` from the same directory, which contains the following listing to specify the names and private IPs of the VMs as shown in Listing 6.3.8.

Listing 6.3.8: Node name and IP confiration for local multiple-VM-node deployment, `cloud-data-platform/src/dep-openstack/salt/servers.yml`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

The `Vagrantfile` also calls `/vagrant/cloudInit/master.sh` and `/vagrant/cloudInit/minion.sh`. Note that the mention of `/vagrant/` here refers to the directory where the mentioning Vagrantfile exists. In terms of the implementation repository structure, this folder is `cloud-data-platform/src/dep-openstack`.

First, consider `/vagrant/cloudInit/master.sh` shown in Listing 6.3.9.

Listing 6.3.9: Bash file to install Salt master for local multiple-VM-node deployment, `cloud-data-platform/src/dep-openstack/cloudInit/master.sh`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

This simply installs salt-master to the VM.

Next, consider `/vagrant/cloudInit/minion.sh` shown in Listing 6.3.10.

Listing 6.3.10: Bash file to install Salt minion for local multiple-VM-node deployment, `cloud-data-platform/src/dep-openstack/cloudInit/minion.sh`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

The above Bash script is called by the Vagrantfile with arguments `master_ip` and `hostname` from the `servers.yml` file. Note that the argument `master_ip` is passed directly via the Vagrantfile to the Bash script script, while `hostname` is collected from the VM that was provisioned with that `host_name` from `servers.yml` by Vagrantfile.

In the sequence of steps described in Sec. 6.3.2.2, the files described above in this section are accessed in the first two commands, identical to those required in Sec. 6.3.1.2.

However, the multiple node deployment requires a further command to be given on the Salt master node of the virtual cluster: The command
`$ sudo salt '*' state.apply` given on the Salt master node calls a `top.sls` file (similar to the one described in Sec. 6.3.1.1) which has been provisioned along with the master node itself according to the Vagrantfile. This means the Salt master has the following file locally as `/srv/salt/top.sls`, which on the repository is stored at `cloud-data-platform/src/dep-openstack/salt/top.sls`, shown in Listing 6.3.11.

Listing 6.3.11: Top Salt file for local multiple-VM-node deployment, `cloud-data-platform/src/dep-openstack/salt/top.sls`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

In the above file, we see the Salt minions are targeted with the hostnames provisioned by Vagrant based on the specifications in `servers.yml`. For example, `zk1`, `elastic3`, and `logstash3`. Just as in the case of a single VM provisioned locally, the salt files describe a sequence of requirements that is enacted on the minions by the Salt master to set the nodes in some state.

Here, the distributed deployment is planned so as to distribute the various functional components of the implementation to avoid capacity bottlenecks and to have fault tolerance.

For example, `elastic1` is part of the Elasticsearch cluster, and should interact with `elastic2` and `elastic3`. We look closer at the salt files involved in applying the `top.sls` state to `elastic1`. These files are `cloud-data-platform/dep-openstack/salt/common/java.sls`, `cloud-data-platform/dep-openstack/salt/common/http_transport.sls`, `cloud-data-platform/dep-openstack/salt/elastic/install.sls`, and `cloud-data-platform/dep-openstack/salt/elastic/createindices.sls`.

We can look at `cloud-data-platform/src/dep-openstack/elastic/install.sls`, shown in Listing 6.3.12.

Listing 6.3.12: Elasticsearch installation Salt file for local multiple-VM-node deployment, `cloud-data-platform/src/dep-openstack/salt/elastic/install.sls`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

Here we see that the prepared Elasticsearch configuration `elasticssearch.yml` shown in Listing 6.3.13 is copied from the repository to `elastic1`.

Listing 6.3.13: Elasticsearch configuration file for local multiple-VM-node deployment, `cloud-data-platform/src/dep-openstack/salt/configs/elasticsearch/elasticsearch.yml`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

Note that `elasticssearch.yml` uses Jinja notation to reference constants stored as Pillar data on the Salt Master. The IPs of the `elastic1`, `elastic2`, and `elastic3` nodes are pre-defined and stored in `cloud-data-platform/src/dep-openstack/pillar/settings_local.sls`, which is accessed via `cloud-data-platform/src/dep-openstack/pillar/top.sls`. Note also that the `pillar` referenced with Jinja notation, e.g. `\{\{ pillar['elastic1'] \}\}`, is put into a default directory on the Salt master node by `Vagrantfile` by the line `c.vm.synced_folder "pillar/", "/srv/pillar"`.

Pillar thus is used to enable access to information that is not otherwise available to the Salt minion. In programming terms, Pillar enables the Salt master to hold and securely broadcast global variables to the minion nodes.

The above example illustrates the sequential steps and great reliance on dependencies in the SaltStack distributed deployment. This makes for a powerful but sensitive deployment tool. Likewise, the `cloud-data-platform/src/dep-openstack/salt/top.sls` file targets the other nodes with specific deployments in the same way, although the details vary.

6.3.2.2 Operations to enact the deployment

In the system where Vagrant is installed, the multi-node deployment can be done by going to the location of the above Vagrantfile and entering the commands shown in Listing 6.3.14 in the command prompt.

Listing 6.3.14: Commands to enact local multiple-VM-node deployment.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

Note that the provision of VMs is taken care of by Vagrantfile in this case, but the distributed deployment of the module implementation happens with an extra step, where the user must manually access the Salt master node and call `/srv/salt/top.sls` by the command
`$ sudo salt '*' state.apply`.

6.3.2.3 Notes on context

In practice, the multiple node deployment is similar to single node deployment, with the primary exception being the need for nodes to be aware of and interact with each other. This challenge can be addressed with the approach of setting the necessary relational configurations as references to Pillar.

Obviously, the variety of nodes dedicated to distinct functions proliferates the number of dedicated salt state files that need to be defined. However, besides managing complex cross-references, this represents only a quantitative increase in the labour required to define the overall cluster deployment state.

To make a simplified summary of dependencies between functional components in the distributed deployment, we note that:
An Elasticsearch node requires a link to the other Elasticsearch nodes, as specified in `elasticsearch.yml`. Logstash nodes needs links to the Elasticsearch cluster and the Kafka cluster. Kafka nodes need links to the Zookeeper cluster. Finally, Acq nodes need links to the Kafka cluster.

Note that this version of deploying a distributed implementation to a virtual cluster provisioned by Vagrant may not be functional or practical due to the scale of the distribution described. However, this section is intended to describe the principle of a distributed deployment. This distributed deployment has been proven in the case of VMs provisioned by OpenStack, as described in Sec 6.4.2. Future development may specify a distributed implementation on a scale feasible in Vagrant.

6.4 Deployments to cloud (Cloud virtual machine environment with OpenStack)

Whereas Vagrant in combination with VirtualBox enabled the provisioning of VMs on a local machine, OpenStack enables the provisioning of VMs in a cloud environment. Since the cloud environment has access to a greater pool of hardware resources, this makes the cloud environment more suited to distributed deployment. Nevertheless, we first describe the single node deployment on a cloud computing platform in Sec. 6.4.1 before discussing the distributed deployment in Sec. 6.4.2.

Note that the steps analogous to those automated by the recipe in a Vagrantfile will, for the cloud environment deployments, partly be described as OpenStack GUI operations in Sec. 6.4.1.2.1, and partly be described as CLI operations in Sec. 6.4.1.2. This can be seen as an instructive illustration and explanation of the process that is otherwise automated. In Sec. 6.4.2, an automation process analogous to Vagrant is described for the distributed deployment to a cloud environment.

6.4.1 Deployment to single VM on cloud

This section describes a single node deployment of the module implementation to a virtual machine hosted on cloud computing platform. The virtual machine that will receive the deployment should have a minimum of 4 GB memory, 10 GB storage, and 4 vCPUs.

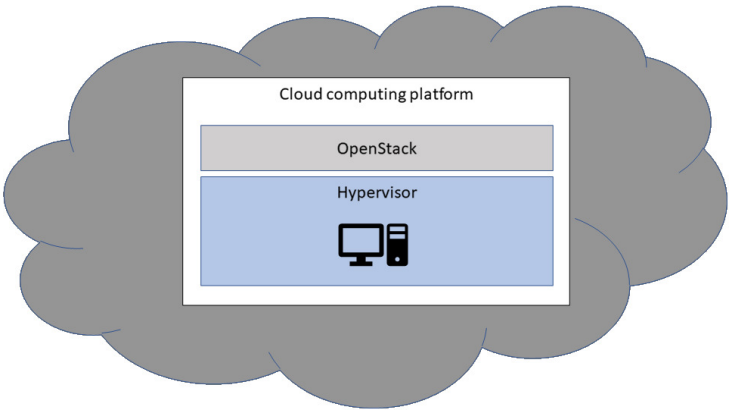

Figure 6.3: Provisioning single VM node on OpenStack.

6.4.1.1 Key files

The files used for deploying the module implementation to a single Virtual Machine on a cloud computing platform are largely the same as in the Vagrant case, in Sec. 6.3.1.

The file `cloud-data-platform/src/dep/salt/prepare.sh`, shown in Listing 6.4.1, is used to copy Salt files and configurations from the cloned repository on the local VM to the default locations expected by the Salt programs.

Listing 6.4.1: Bash file for preparing single VM-node deployment on cloud, `cloud-data-platform/src/dep/salt/prepare.sh`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

6.4.1.2 Operations to enact the deployment

Before the deployment process itself can be executed, an appropriate target for the deployment must be provisioned. This provisioning is explained in Sec. 6.4.1.2.1 before the deployment to the resulting VM is described in Sec. 6.4.1.2.2.

6.4.1.2.1 GUI operations

OpenStack Dashboard GUI is used to provision the single VM for this deployment. Click `Launch Instance` and follow the dialog box to specify the details of the VM to be instantiated. The virtual machines that will receive the deployment should have a minimum of 4 GB memory, 10 GB storage, and 4 vCPUs.

To deploy the cloud data platform on the cloud computing platform, first the necessary virtual machines must be provisioned. The multiple node deployment requires multiple VMs, but the steps for each VM are essentially the same as in Sec. 6.4.1.2.1, although the details vary for each type of VM depending on its intended purpose in the overall virtual cluster.

6.4.1.2.2 CLI operations

The following commands will deploy the module implementation on a virtual machine. Remember to SSH into the VM in question first, with a SUDOer user.

Make sure that the VM hostname is set appropriately in the local file `/etc/hosts` so that the first line looks as shown:

```
127.0.0.1 localhost cdp-single
```

Here the `cdp-single` is the name of the example VM used throughout this section. `cdp-single` is the hostname of the VM set in the cloud computing platform interface when provisioning the VM.

Next, install Salt master and minion on the VM with the CLI operations as shown in Listing 6.4.2.

Listing 6.4.2: Commands to enact single-VM-node deployment on cloud.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

Now, establish a link between Salt master and Salt minion by editing the local file: `/etc/salt/minion` and uncommenting and modifying the variable `master`. Check the default value by the following command, and look at the first line returned as shown in Listing 6.4.3.

Listing 6.4.3: Check Salt minion configuration file for single-VM-node deployment on cloud.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

Modify `/etc/salt/minion` with an editor like Vi or Nano so that repeating the above command gives the following result as shown in Listing 6.4.4.

Listing 6.4.4: Edit Salt minion configuration file to link with Salt master for single-VM-node deployment on cloud.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

To make the Salt minion service take into account the above edit and recognize the localhost as the Salt master, enter the following command:

```
ubuntu@cdp-single:~$ sudo service salt-minion restart
```

The command shown in Listing 6.4.5 returns the state of acceptance of Salt minion SSH public keys submitted to the Salt master.

Listing 6.4.5: Check Salt minion status for single-VM-node deployment on cloud.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

As seen in the response to the above command, the `cdp-single` Salt minion keys have not been accepted by the Salt master.

To accept the `cdp-single` key, run this command:

```
ubuntu@cdp-single:~$ sudo salt-key -a "cdp-single"
```

Note that, in the case of multiple Salt minion keys pending acceptance, the command `$ sudo salt-key -A` would accept all pending keys.

Test that communcation is established between the Salt minion and the Salt master by running the command shown in Listing 6.4.6 and ignoring any warnings raised.

Listing 6.4.6: Verify Salt minion status for single-VM-node deployment on cloud.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

The response to the command should be as above. Note that the `sudo salt` part of the command instructs the Salt master to address its cluster of Salt minions. Given the `'*'` term, all connected Salt minions should respond.

Next, generate a SSH key pair with blank password and store it in the default file `'~/home/ubuntu/.ssh/id_rsa'`, where `ubuntu` is the username of the relevant SUDOer on the VM. Use the command shown in Listing 6.4.7.

Listing 6.4.7: Generate SSH key for single-VM-node deployment on cloud.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

This key will be used to clone the `cloud-data-platform` repository.

Logging in on `www.gitlab.com`, click on the `triangulum/cloud-data-platform` project, then on the left-hand menu on "Settings". This will drop down a further menu. Click "Repository". Scroll down on the resulting page to a section titled"Deploy Keys" and click the button labelled "Expand".

Under "Create a new deploy key for this project", enter as title the VM name. In this example, "cdp-single".

Copy the contents of `/home/ubuntu/.ssh/id_rsa.pub` into the field "Key" under the same heading. Click "Add Key". (Write access should be assigned only sparingly to keys.)

Now, clone the repository to the VM using the following command:

```
ubuntu@cdp-single:~$
git clone git@gitlab.com:triangulum-cdp/cloud-data-platform.git
```

Now, prepare the node before Salt deployment with a Bash script that will copy the necessary files to `/srv/salt/`:

```
ubuntu@cdp-single:~$ cd cloud-data-platform/src/dep/salt/
ubuntu@cdp-single:
~/cloud-data-platform/src/dep/salt$ sudo ./prepare.sh
```

The `prepare.sh` file is discussed in Sec. 6.4.1.1.

Finally, deploy the module implementation using Salt by the following commands:

```
ubuntu@cdp-single:~/cloud-data-platform/src/dep/salt$
cd /srv/salt/
ubuntu@cdp-single:/srv/salt$ sudo salt '*' state.apply -t 600
```

There may be a problem with the previous command, in which case the response lists in the summary a non-zero number of failed states. Scrolling up through the more verbose parts of the response, the particular failures may be troubleshot.

If the installation of the package `cdpadaptor` has failed, but the repository has been cloned locally, the following commands can fix the deployment:

```
ubuntu@cdp-single:/srv/salt$ cd /home/ubuntu/cddadaptor
ubuntu@cdp-single:~/cddadaptor$ sudo -H pip3 install .
```

Check that the `cdpadaptor` package was installed with this command:

Information about cdpadaptor python package

```
ubuntu@cdp-single:~/cddadaptor$ pip3 show cdpadaptor
---
Metadata-Version: 1.1
Name: cdpadaptor
Version: 0.0.0
Summary: Superclasses for adaptors used in the
 Cloud Data Platform to acquire data
 and forward it to Kafka.
Home-page: UNKNOWN
Author: UNKNOWN
Author-email: UNKNOWN
License: UNKNOWN
Location: /usr/local/lib/python3.5/dist-packages
Requires: kafka, prompt-toolkit, PyYAML
Classifiers:
  Programming Language :: Python
  Programming Language :: Python :: 3.6
  Programming Language :: Python :: Implementation::CPython
```

Now, re-apply the Salt state. The same failure may be reported, but the dependencies will be satisfied with the above fix:

```
ubuntu@cdp-single:~/cddadaptor$  sudo salt '*' state.apply -t 600
```

The various components of the module implementation should now be up and running. The various services can be checked with commands such as:

```
$ sudo systemctl status elasticsearch.service
```

The system should now be running, a python process running the `kolumbus_vm_client_tcp.py` collects data from the Kolumbus VM (Vehicle Monitoring) interface once every 60 seconds, this starts automatically when state.apply is run. This data is passed to the systemd service logstash, which preprocesses the data and store it to the systemd service elasticsearch. The systemd process kibana exposes the Kibana web interface for data visualisation and exploration though port 8080. If the VM setup forwards some port to 8080 on the guest machine the Kibana interface will be available from a browser on the host machine.

6.4.1.3 Notes on context

Deployment to a single node on cloud is the production version of single node deployment using Vagrant, as described in Sec. 6.3.1. The differences lie in how to create a virtual machine and how to bootstrap SaltStack. Once Salt master and Salt minion are sorted out, the rest of the deployment procedure is very much the same as with Vagrant.

6.4.2 Deployment to multiple VMs on cloud

This section describes a multiple node deployment of the module implementation to a set of virtual machines hosted on cloud computing platform. The VMs to host the various components may have different performance requirements depending on the functions of the hosted components.

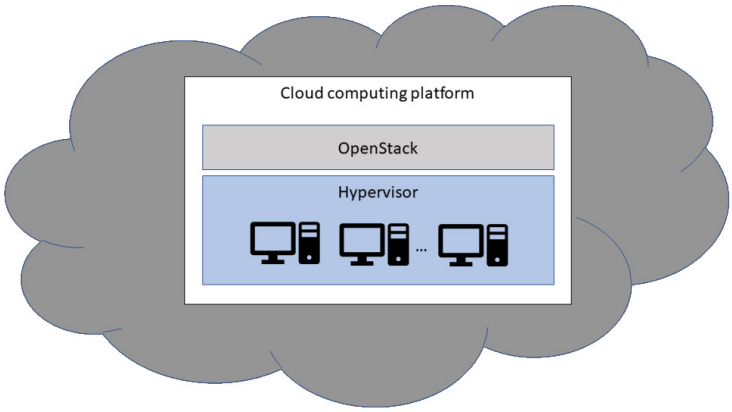

Figure 6.4: Provisioning multiple VMs node on OpenStack.

6.4.2.1 Key files

The files used for deploying the module implementation to multiple virtual machines on a cloud computing platform are largely the same as in the multiple virtual machines with Vagrant case, in Sec. 6.3.2.

The prose descriptions in this subsection will use file paths relative to the repository directory `cloud-data-platform/src/dep-openstack`, since this directory contains all the necessary files for the deployment to multiple nodes provisioned on a cloud computing platform. For example, this subsection may discuss the files `salt/top.sls`, and in all prose descriptions, such paths will be given relative to `cloud-data-platform/src/dep-openstack`. This is done both for brevity and clarity.

The "Quick Start" guide to deploying this deployment scheme is described in `cloud-data-platform/src/dep-openstack/Readme.md`.

In the case of multiple virtual machines using Vagrant (in Sec. 6.3.2), the virtual machines are created programmatically according to the `Vagrantfile`. The analogous operations of creating virtual machines on the cloud computing platform is done using the OpenStack Dashboard GUI. During the creation process of virtual machines, it is possible to include a Bash script that will be executed when the virtual machine boots for the first time. In this deployment scheme, the script will either set up a Salt master node using the script `master.sh` shown in Listing 6.4.8, or a Salt minion node using the script `uis-init-minion.sh` shown in Listing 6.4.9. The process of installing Salt master must be executed for at least one virtual machine with the hostname `salt-master`. In contrast, the process of installing salt-minion must be executed for every virtual machine on the cluster – 16 in the present case.

Listing 6.4.8: Bash file for installing Salt master for multiple-VM-node deployment on cloud, `cloud-data-platform/src/dep-openstack/cloudInit/master.sh`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

Listing 6.4.9: Bash file for installing Salt minion for multiple-VM-node deployment on cloud, `cloud-data-platform/src/dep-openstack/cloudInit/uis-init-minion.sh`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

As a result, the created virtual machines will have the minimum required software packages, making it ready to join the deployment cluster of `salt-master`.

In addition, it is important to have the file `cloud-data-platform/src/dep-openstack/pillar/settingscloud.sls` consistent with the virtual machine instances. This file is basically a mapping between an instance of CPD architecture and a set of `salt-minion` instances.

For example, if a VM with `ip:192.168.1.51` exists as a `salt-minion`, and you want that virtual machine to act as one of the `kafka` nodes, say `kafka1`, then you have to make sure that the key `kafka1:` in `cloud-data-platform/src/settingscloud.sls` as shown in Listing 6.4.10 gets the right `ip` (`kafka1: 192.168.1.51`).

Listing 6.4.10: IPs for subclusters and software versions are set in `cloud-data-platform/src/dep-openstack/pillar/settingscloud.sls`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

6.4.2.2 Operations to enact the deployment

Before the deployment process itself can be executed, the appropriate targets for the deployment must be provisioned. Using the OpenStack GUI on CCP, provision the appropriate number of VMs and make sure the associated computing and network resource capacities are adequate. Also, make sure the intended target IPs of the newly provisioned VMs match those indicated in the `settingscloud.sls` shown in Listing 6.4.10. This may be best done by editing the file after provisioning the target VMs and reading off the IPs provided by OpenStack.

Having accomplished the VM provisioning above, proceed to install salt minion and master on the appropriate VMs, clone the `cloud-data-platform` repository to each VM, and then:

1. ssh into salt-master.
2. List minions: `sudo salt-key --list-all`.
3. Accept the salt-minions: `sudo salt-key --list-all`.
4. Check that the file `settingscloud.sls` is consistent with the salt-minions.
5. Clone the git repository: see Sec. 6.4.1.2.
6. Run the script under `cloud-data-platform/src/dep-openstack/salt/prepare.sh`.
7. Run `sudo salt '*' state.apply -t 600`.
8. From OpenStack GUI associate a floating ip to `Kibana` node.
9. Open a web-browser and navigate to (Kibana node IP):(port).
10. The cloud data platform's accumulating data collection can then be seen via the Kibana GUI.

6.4.2.3 Notes on context

Deploying CDP cluster on OpenStack is the production prototype analog of multiple VMs using Vagrant. The differences from a software development perspective are mainly on how to create, provision and connect to the virtual machines. The SaltStack part is exactly the same.

To take advantage of the data processing framework beyond, additional VMs should be provisioned and the deployment under `cloud-data-platform/src/jupyspark` should be enacted on the additional VMs to form a combined JupyterHub and Spark cluster. This is not appropriate unless the cloud environment can provision additional VMs fully independent from the multiple-node data collection framework deployed based on `cloud-data-platform/src/dep-openstack` or a similar deployment.

Based on the ample examples provided by the above code and command listings, the `cloud-data-platform/src/jupyspark` deployment should be relatively simple, running first a Bash file `cloud-data-platform/src/jupyspark/salt/prepare.sh` and then applying the usual salt state as coordinated via `cloud-data-platform/src/jupyspark/salt/top.sls`. Note that minor configuration updates may be necessary, depending on VMs provisioning, to correctly reflect the nodes' local hostnames and IPs.

6.5 Alternative deployment schemes

This section discusses necessary considerations for making custom deployment schemes for the module implementation.

Note that Sec. 6.4.2 describes a particular deployment scheme on a pre-defined set of VMs. That deployment scheme represents a reasonable distributed deployment within some arbitrary resource restrictions. Such restrictions are reasonable from a economic perspective, i.e. to promote efficient resource use in the development of the deployment scheme.

However, alternative deployment schemes on a cloud computing platform should be informed by the available resource pool. Resource availability may vary greatly, although it should be determined and fixed before the deployment scheme development begins. Most importantly, the resource pool should not shrink below what the deployment scheme is developed to require.

Future developments of the implementation and associated deployment schemes are encouraged to take note of the division of labour exemplified in Sec. 6.4.2. Subsets of the virtual cluster were dedicated to specific functions, typically ensuring fault-tolerance for the function in question by distributing each function with multiple nodes. For example, Elasticsearch was distributed to multiple dedicated nodes, with no overlap with the set of multiple nodes dedicated to Kafka. These subsets of the overall virtual cluster, each of which are dedicated to a specific function, can be called "functional groups." Not only does this ensure fault-tolerance within the functional group, but it isolates the different functions of the module implementation from one another, preventing one function from impinging on the resource requirements of another.

In principle, the various functional groups of the modules are separately scalable. However, Sec. 6.3.2 and Sec. 6.4.2 describe multiple node deployments where the number of nodes dedicated to each functional group is limited and pre-defined. The primary challenges include planning an optimal allocation of available resources to the various components in the module implementation, thus defining the size of the corresponding functional groups, as well as updating the dependencies in configuration files that enable coordination within and across functional groups.

Accordingly, an alternative deployment scheme can be established by a few simple steps, starting with `pillar/settingscloud.sls`. We are here continuing the relative file path convention in Sec. 6.4.2, i.e. refering to file paths in the repository that are relative to `cloud-data-platform/src/dep-openstack`. First of all, edit the list of hostnames and IPs so that the hostname consists of the name of the functional group and an integer in a sequence, e.g. `elastic2` for the second node in the Elasticsearch cluster. This enables independently scaling up or down the different functional groups in the overall virtual cluster. Second, make sure that the files that depend on the variables broadcast by Pillar are updated to listen for the correct, new number of nodes in each functional group.

For example, the adaptors implemented with `cdpadaptor` rely on a file `/salt/configs/acq/kafka_brokers.json` which is copied to a local file `/tmp/kafka_brokers.json` on the VMs where the adaptors are deployed. The contents are shown in Listing 6.5.1.

Listing 6.5.1: Information to connect adaptor VMs to the Kafka cluster, `/tmp/kafka_brokers.json`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

This file (or its analog) must be updated to reflect changes made in `pillar/settingscloud.sls` (or its analog) in an alternative deployment scheme.

Thus, if the new deployment should have four nodes in the Kafka functional group, `pillar/settingscloud.sls` should contain a line below `kafka3`, for example as shown in Listing 6.5.2.

Listing 6.5.2: Modifying a Pillar file for alternative deployment, `.../alternative/pillar/settingscloud.sls`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

Then, `kafka_brokers.json` should become modified as as shown in Listing 6.5.3.

Listing 6.5.3: Information to connect adaptor VMs to the modified Kafka cluster, `/tmp/kafka_brokers.json`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

The above example illustrates how to manipulate the existing multiple nodes deployment scheme under `cloud-data-platform/src/dep-openstack` to configure alternative deployments. Note that a variety of files rely on the variables broadcast by Pillar based on `pillar/settingscloud.sls`, and these files may need to be updated to reflect the alterations desired.

6.5.1 Ideas for possible alternative deployment schemes

Note that data access could conceivably be supported as a separate functional group by deploying the Elasticsearch cluster with some nodes dedicated to simply mirroring and making available the stored data, while other Elasticsearch nodes would be dedicated to storing the data incoming from data ingest nodes.

Chapter 7

Replication Guide

7.1 Overview

The purpose of this chapter is to advise technical personnel who may wish to replicate the UiS module implementation in their own allocation environment. As practical guidance, the chapter is intended to be comparatively short and easy to navigate, with references to more dense material in earlier chapters. Replication may certainly involve modification of the module implementation described in this report, and this is hopefully encouraged by the documentation represented by Chapters 2 through 6, but initially it may be informative and useful to simply deploy the solutions as presented to gain familiarity with the components and their relationships.

Sections 7.2 through 7.4 qualitatively discuss different scenarios that may be relevant for replicating parties in Follower Cities or elsewhere.

Note that to gain access to the repository before it is made public, contact Triangulum researchers at UiS, identifying yourself, asking for repository read-access, provide credentials as a consortium member representative or an appropriate representative of the EU commission services, and the public key of the machine to which a repository copy is desired. To get a public key with a Linux machine, follow the instructions in Sec. 6.4.1.2.2, particularly those shown in Listing 6.4.7.

7.2 Demo implementation with vagrant and single node

If the organization is not already committed to replicating the UiS module implementation, it may be reasonable to take the time to become practically familiar with the system on a local virtual machine, as discussed in Sec. 6.3.1. This approach uses source code in the`cloud-data-platform/src/dep` and `cloud-data-platform/src/vagrant` directories. This should show how to deploy the data collection framework, and the Kibana part of the data analytics toolkit.

This will enable those interested in replication to try out the solution with minimal hardware before committing to a more serious investment in hardware infrastructure. This will also enable technical personnel to get an idea of the practical possibilities

and requirements to give decision-makers a relevant overview before more resources are dedicated to a fuller replication.

7.3 Distributed on hardware

An alternative or next step to the very simple deployment recommended in Sec. 7.2 is to set up a distributed deployment on a cluster of physical machine, where the network and computing resources are not virtualized. This is called "running on bare metal." This may again save on resource commitment, avoiding either the building of a data centre and a cloud computing platform, such as CCP, but can provide real capacity beyond the toy-replication of Sec. 7.2.

On the other hand, this deployment approach would require a more hands-on approach to adapt the deployment configured under `cloud-data-platform/src/dep` and `cloud-data-platform/src/dep-openstack` to the allocation environment represented by the physical machines and network.

However, unlike the single local VM deployment described in the previous section, deploying to a physical cluster would potentially mean that the part of the data processing framework/data analytics toolkit configured in the source code under `cloud-data-platform/src/jupyspark` could be deployed as well. Deploying `jupyspark` to a single machine, especially a single virtual machine, will not give any great advantage over simply installing a single instance of Apache Spark and Jupyter Notebook locally. However, with multiple machines, the power of distributed computing can be harnessed, and the `jupyspark` source code facilitates such deployment.

7.4 Cloud hosted

Finally, a full deployment of `cloud-data-platform/src/dep-openstack` and `jupyspark` can be achieved if the allocation environment is a cloud computing platform with adequate resources. This is in principle almost as simple to do as Sec. 7.2, once the provisioning of VMs is done, and the correct IPs are configured as discussed in Sec. 6.5.

However, the configuration of the various component clusters should be adapted to the requirements of the module replication, and must be undertaken as an iterative development by the replicating organization. Note that commercial cloud hosting services may be appropriate, but this depends on the requirements of the replicating party. Scalable costs and legal liabilities with respect to hosting data remotely should be considered before choosing such an option.

An intermediate option is to deploy the cloud data platform to a single VM on the cloud, as opposed to on a VM provisioned locally as discussed in Sec. 7.2. However, if the local deployment has been undertaken, and cloud deployment is well understood, this is not a necessary step pedagogically speaking.

7.5 New data sources, new adaptors

Since a replication of the UiS module implementation by Follower Cities or others will not require collecting data from the same data sources as module 544 cloud data platform supporting WP2 impact reporting, the replication will require new data sources, and therefore new adaptors, to be useful. To generate each new adaptor, the `cloud-data-platform/generator/cdpadgenerator.py` script can be used after installing `cdpadaptor` package for Python.

Ensure that the machine's public key is a deploy key at the `cdpadaptor` repository. Next clone the git repository to some local directory, then use the `pip` package management tool to install the package and make it available system wide, as shown in Listing 7.5.1.

Listing 7.5.1: CLI commands to install `cdpadaptor`.

```
$ git clone git@gitlab.com:triangulum-cdp/cdpadaptor.git
$ cd cdpadaptor
$ sudo -H pip3 install .
```

The new adaptor can then be generated with `cloud-data-platform/generator/cdpadgenerator.py`. This generates the adaptor Python script, Systemd service (and timer if appropriate), Logstash pipeline configuration files, and Salt files. The data transfer methods for the data source need to be well understood to do this step correctly. The generated files then have to be appropriately added to a deployment directory under `cloud-data-platform/src/`, pushed to a repository under the replicating party's control, and pulled by the machines which should deploy the new adaptor using Salt. Make sure the Salt file generated for the adaptor is added to the `top.sls` for the deployment in question. Finally, from the Salt master deploy the new Salt state to the Salt cluster.

Chapter 8

Data analytics use case: Traffic flow analysis

As part of the UiS module implementation, an exemplary application of the module 542 data analytics toolkit, or use case, was commenced. The implementation of the data processing framework included both Kibana and JupyterHub with Spark, via the `jupyspark` deployment on CCP. This chapter provides a brief summary of this effort as it currently stands.

In mathematics and civil engineering, traffic flow is the study of interactions between travellers (including pedestrians, cyclists, drivers and their vehicles) and infrastructure (including highways, and traffic control devices), with the aim of understanding and developing an optimal transport network with efficient movement of traffic and minimal traffic congestion problems.

In this case study, the ultimate intention is to create analytics on traffic flow in Stavanger and the surrounding region in Rogaland county. In particular, Triangulum consortium member Kolumbus has been providing open data on bus transport, which has been collected by the data collection framework in the cloud data platform. This data has subsequently been made available to the data processing framework. The intention of the use case analysis effort is to explore the data, to understand its underlying structures, and to make predictive models on when busses arrive at bus stops while in service.

Ultimately, it may be possible to run the predictive model as a public service, providing bus arrival time prognoses with better accuracy than the current prognosis system. Second, perhaps insights from this effort can help optimize route planning for Kolumbus and members of the public using private transport.

For this use case, the primary data source type was the Kolumbus real time open data, which is standardized with the Service Interface for Real Time Information (SIRI) protocol.

To infer the structure of the traffic network from the data source is a natural first step, as this should provide a static background for the dynamics of vehicles moving through space and time. Furthermore, it is noted that the data source implicitly assumes a graph structure in its traffic network, and mapping this may be powerfully enabling to subsequent steps.

The SIRI VM (vehicle monitoring) data source for Kolumbus was the most consistently

available and informative data source for the purposes of the use case. Crucially, the Kolumbus VM data specifies GPS positions together with timestamps, as well as the status of a monitored bus with respect to the bus route or line the the bus is serving at the time. This data also includes which bus stop the bus is currently approaching.

To model a bus network using graph theory, we can build a graph where the vertices represent bus stops. Then the edges of the graph represent paths taken by busses that connect the bus stops as nearest neighbours on a given bus route. Mathematically, the traffic network can then be represented by an adjacency matrix. In the simplest case, the adjacency matrix would represent the nearest neighbour bus stops as a 1 in the matrix where the bus stops are represented as the row and column number. The adjacency matrix is sparse, and thus all other elements are 0. Thus, the adjacency matrix would be symmetric.

However, the matrix elements representing the edges of the traffic network graph model could be weighted in some way to reflect properties of the traffic network being modelled. For example, the density of traffic flow could be represented by giving the adjacency matrix elements variable scalar values. In this formulation, the adjacency matrix is not guaranteed to be symmetric. Another weighting of the adjacency matrix elements could be the distance between the bus stops, or (some statistical aggregate of) the distance travelled by busses between the two stops. Since roads are not straight lines, the latter category is highly relevant.

Other graph theoretic properties are also interesting to explore, such as total traffic inflow/outflow for each bus stop, centrality of the bus stops in the network.

We can define a network (graph) as a system of nodes (vertices), with links (edges) belonging to a single modal type. In this definition, a node, like a bus stop, is a location on a transportation route that has capacity to generate traffic flow, while a link, like a road, is a connection between two nodes along which flow occurs.

8.1 Implementation of the graph

To create the graph of buses, it was necessary to discover the main elements of the graph from the data source including vertices (bus stops) and edges (roads or paths between nearest neighbour bus stops). Both the vertices and edges of the graph could not be directly read from the Kolumbus VM data. However, `http://sanntidsappservice-web-prod.azurewebsites.net/busstops` provided an overview of the bus stop names, their numerical identifiers in the SIRI protocol as implemented by Kolumbus across their multiple monitoring services, and the nominal GPS position of each bus stop.

Given the time series data with minute-by-minute position histories for each bus (when monitored, which was often not the case in the data set), it was clearly feasible to parse out the subset of history that traced the path of a given bus traversing a given route, and hence infer the sequence of bus stops on that route. At this stage, the nominal route was not a priority compared to developing the ability to discover the empirical route taken by a bus.

To develop this approach, the next step was to separate out a workable subset of the bus network based on a single bus line, which could be served by multiple buses.

8.2 Technical steps in preparing the analysis

On a Jupyter Notebook session provided by JupyterHub, with a PySpark session kernel, we imported required packages from PySpark and PySpark.sql to read data with the format of Spark abstractions like RDDs (Resilient Distributed Dataset) and DataFrames, respectively.

As the Kolumbus VM data is indexed and stored in an Elasticsearch cluster in the data collection framework, the data could be read via the Elasticsearch-Hadoop connector deployed along with the rest of the `jupyspark` deployment.

To work only with the relevant subset of the bus network, the Elasticsearch query used to extract data from storage was confined to a single `LineRef` value in the Kolumbus VM data set. The query also filtered out the data points (indexed documents) where the vehicles (buses) in question had for any reason not been monitored at the time of the data point. The query furthermore filtered out data points lacking the bus stop code (`StopPointRef`) identifying the next stop for the bus on its current trip. Finally, the query filtered out the data points the bus stop code (`OriginRef`) of the origin terminal of the current trip.

Having extracted this tractable subset of the data and loaded it to an RDD, the next step was to create from it a DataFrame, to support more convenient operations on the data subset. The read data from Elasticsearch is first read as a pySpark RDD, containing Row objects. Each object includes a unique identifier of the record generated by the indexing in Elastiscearch, and a series of information about the bus status and positioning. These two parts are formatted in a tuple with string and nested dictionary data types, respectively. A list of these tuples forms the whole RDD.

In order to make a DataFrame out of this RDD object, we can use Spark SQL. Spark SQL can convert an RDD of Row objects to a DataFrame, inferring the data types. In this way, rows are constructed by passing a list of key/value pairs as key-word arguments to the Row class. The keys of this list define the column names of the table, and the types are inferred by sampling the whole dataset.

On the other hand, when a list of key-word arguments cannot be defined directly ahead of time, as in the present case, where there is a list of tuples including nested dictionaries, a DataFrame can be created programmatically with three steps.

1. Create an RDD of lists (or tuples) from the original RDD; for this purpose, a function was written to read the data in the original format mentioned above and convert it to an RDD in which each Row object contains only a flat dictionary, instead of a nested one, with all the information of the read tuple in the format of keys and values. The keys would be the column names and values would be the column values in that row.

2. Create the schema represented by a StructType matching the structure of lists in the RDD created in step 1.

3. Apply the schema to the RDD via createDataFrame method provided by SparkSession object.

In the second approach that was used to create a DataFrame, the schema is already available, thus the process of creating the DataFrame was simplified, compared to the first approach where data types (schema) are inferred by sampling the whole data set.

8.2.1 Transformations on time-related DataFrame columns

Having extracted relevant data and transformed it to an amenable data structure, the DataFrame was required to be ordered by time. Since the bus trips occur in a sequence of time, the sequence of positions occupied by the bus on a trip depend on time ordering. Several columns in the DataFrame included time information about data, such as scheduled and expected departure times from the origin terminal stop or similarly expected and scheduled arrival time to the destination terminal stop of a bus trip.

The values of such columns had first been coded with `string` data type, which cannot be programmatically ordered or filtered based on time, and hence needed to be converted to valid datatypes for this purpose, such as `timestamp` or `datetime`.

To solve this issue, a function was defined which converts variables with `string` data type to variables with `datetime` type and assigns them to new columns in the DataFrame. These new columns can be defined with similar names to the original columns, but once the type conversion is found reliable, it is more convenient and less confusing to drop from the DataFrame columns with time data in `string` type.

8.3 Defining and running an algorithm to find the vertices and edges of the subgraph

To make the intended a graph, an algorithm was defined which can iterate through all the journeys that buses associated with the selected line reference have experienced during the period constrained by the data collection implementation. This period, which lasts about two years, corresponds to the time when the real time data from Kolumbus VM has been collected to the data collection framework.

Each data point is primarily anchored in time by the field `RecordedAtTime`, and the DataFrame was ordered along this column. The algorithm to infer the graph can be summarized as follows:

1. Define a new DataFrame derived from the original DataFrame but with only the columns that are of interest for recognizing the elements of the graph.

2. Filter out the records where there are `None` or `Null` values for the specified columns. Columns that were important for data quality and should not be empty included `JourneyRef`, `OriginRef`, and `DestinationRef`.

3. Duplicate records with the same values for both `VehicleRef` AND `RecordedAtTime` should also be filtered out.

4. Add new columns to the new DataFrame and initialize these to `0`, `None` or `False`. These default values can subsequently be overriden by the inferring algorithm. These

new columns with values to be inferred later included “PreviousBusStop”, “Stop counter”, and “Edge”, the latter of which should consist of a tuple of the previous and next bus stop codes.

5. Iterate through each journey that has occurred in the specified bus line. At each journey each record (observation) should be compared with the previous record in terms of some fields (columns) like the `StopPointRef` field representing the stop code for the next upcoming bus stop on the current bus trip.

By iterating through the chronological sequence of records within a given bus journey history, a number of edges between nearest-neighbour bus stops can be inferred. This set of edges can subsequently be mapped to an adjacency matrix. This shows how the same process can be undertaken for all the `LineRef` values in the data set, to build up a comprehensive adjacency matrix for the underlying graph of the traffic network.

8.4 Summary of Analytics Use Case

Subsequent analysis efforts have been started using `geopy.distance` and `pandas` dataframe. To take full advantage of the distributed computing, the execution of inferring an adjacency matrix from the complete Kolumbus VM data set should likely be parallelized based on the separate `LineRef` values.

The immediate challenge in the ongoing exploratory analysis of this use case consists in developing various measures of the distance travelled by busses between nearest neighbour bus stops. With `geopy.distance`, it is possible to calculate a geodesic distance (i.e. the shortest path along a curvature model of the Earth) between GPS coordinates. While this is unlikely to correspond reliably with the distance travelled by busses along the roads connecting two nearest-neighbour bus stops, it is an interesting variable to keep track of, as a basis for comparison.

Furthermore, the geodesic distance between different bus positions and the bus stops the bus is moving between at any given time can help to characterize the distance travelled along the road. Interestingly, the Kolumbus VM data model includes an element called `ProgressBetweenStops` with sub-elements `LinkDistance` and `Percentage`. Here `LinkDistance` is the number of meters travelled by the bus from the previous bus stop, and `Percentage` reflects the corresponding percentage of progress along the edge of the graph.

Thus, a `Percentage` of 100% represents the full distance of the edge, and consequently the road distance the bus must travel between two nearest-neighbour bus stops. Some spot checks were made to ensure that the 100% `LinkDistance` was consistent along a given edge for all records with non-zero `LinkDistance` and `Percentage` values. This appears to hold true.

However, the `LinkDistance` and `LinkDistance` values did not appear to consistently reflect the geodesic distance between bus and the next bus stop, which makes sense since the roads are not straight. It remains to be verified that the nominal `LinkDistance` values for the edges on the graph consistently reflect the distance of the paths travelled by the busses.

As further data quality checks and refinements are made, as well as accumulating data, it should be possible to accurately and precisely plot the paths taken by the busses on a map, to better account for irregularities in a given bus trip in terms of the time taken or the path taken. Together, these inferences begin to lay the foundation for improving upon the commercial prognosis application currently used by Kolumbus to inform the public of transport availability.

Furthermore, while the first challenge is to emulate the Kalman filter based solution currently in use to establish a well-understood benchmark model, alternative machine learning algorithms can be tested on the data set to generate competitive models, as well as taking additional data sets, such as weather data, into account. Ultimately, robustness to issues regarding data quality and data completeness is likely to be a key characteristic of a competitive alternative machine learning algorithm for this use case.

Chapter 9

Conclusion

The UiS module implementation has been presented in a fair level of detail, including the motivating and constraining background provided by the Triangulum project itself and the current state of the art in comparable projects. The rationale behind the overall system architecture and the lower level design decisions has been presented to show the layers of abstraction and separation of concerns. The details of implementation and deployment have been presented to both give insight for direct replication, and to provide a basis for further development. After a detailed presentation of the system as a whole, high-level guidance was presented to make the arguments and facts already presented into a set of plausible scenarios with recommended approaches for each. A use case for the data analytics toolkit was described in its current state of development, including the remaining open questions and the ambitions beyond the current point.

The module implementation as a whole has shown its applicability for a variety of tasks within the Triangulum context, and beyond. While data collection and analysis were accomplished within a big data paradigm, the experience of the development has highlighted the challenge of communicating about data across domains of expertise, as well as of clarifying the liabilities associated with data sharing between organizations to lower the threshold enough do so. Finally, human resource allocation is a key challenge in a comprehensive ICT development of the scope of the UiS module implementation.

Possible future directions of development in the UiS module implementation within Triangulum may include various types of improvement.

For the sake of replication, a public code repository may be created that is clean of details regarding consortium members and their modules. This would support replication by not requiring registration to access a closed repository.

If a meaningful function can be defined, it may be interesting to create a read-only dynamic visualization dashboard for modules or Lighthouse Cities. Note that a challenge here is finding a way to combine data sets that is not merely contrived. Another challenge in this regard is the shortage of real time data sources. While several modules can provide time series data, very few can do so in real time, and generally the diverse data sources may not be meaningfully related to one another.

The data analytics use case described in Chapter 8 can be developed further, and by prioritzing this avenue of development, the data processing framework itself will inevitably be further developed.

Both for the visualization dashboard, for the bus arrival time prognosis service, and for offering registered external uses controlled access to the data processing framework, development of external access mechanisms would require additional developments.

Finally, the cloud data platform and its elements will require ongoing maintenance and upgrades to continue functioning nominally in the constant technological evolution seen by networked systems. Even re-design of certain components may become warranted as the UiS module implementation continues supporting the impact reporting of WP2.

Glossary

agreement problem Also called Consensus problem, the problem of making multiple distributed processes maintain some common state or to decide on a future action. 14, 15

big data A term for data that is quantitatively large in terms of volume, velocity, and variety. 5, 11, 14, 28, 34

cloud computing A paradigm of ICT concerned with shared pools of computing resources served over networks to remote clients. 14

dependable computing The field of study focussed on computing systems that can be relied upon. 15

distributed computing A field of computer science concerned with distributed systems that can perform coordinated functions. 14

fault tolerance The property of a system to tolerate failures and still continue to function to some extent. 15

Python A popular programming language with broad support in the open source community. 34, 39, 42, 44, 54, 57, 58, 83, 84

Triangulum An EU-funded Horizon 2020 project on smart cities. 5, 6, 10, 11, 14, 17, 18, 19, 20, 21, 23, 24, 26, 27, 28, 30, 34, 36, 38, 43, 82, 85, 91

Acronyms

API Application programming interface, the set of definitions that enable programmable communication between software components.. 30, 34, 38, 39, 41, 42, 43, 44, 45, 46, 51, 57, 58

CCP The physical data centre and software environment for provisioning virtualized resources for cloud computing applications at UiS. 11, 26, 27, 28, 33, 35, 37, 55, 60, 78, 83, 85

EU European Union. 5, 26, 82

GUI Graphical user interface. 38, 57

ICT information and communications technology. 5, 10, 14, 17

IoT Internet of Things. 10, 11, 14

IRIS International Research Institute of Stavanger. 6

SIRI Service Interface for Real Time Information, a protocol for monitoring public transport. 85, 86

UiS University of Stavanger. 5, 6, 10, 11, 12, 14, 17, 22, 23, 24, 26, 27, 28, 30, 32, 33, 34, 36, 37, 38, 41, 43, 51, 52, 57, 82, 83, 85, 91, 92

vCPU Virtual central processing unit. 27, 71

VM Virtual machine. 27, 33, 37, 52, 56, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 83

Bibliography

[1] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*, 2nd ed. Springer Publishing Company, Incorporated, 2011.

[2] L. Lamport, "Paxos made simple," pp. 51–58, December 2001. [Online]. Available: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[3] The IEEE Technical Committee on Dependable Computing and Fault Tolerance – IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance. Dependable computing and fault tolerance. Accessed: 2018-01-12. [Online]. Available: http://www.dependability.org/

[4] Wikipedia, "Critical system — wikipedia, the free encyclopedia," 2017, [Online; accessed 6-December-2017]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Critical_system&oldid=790821273

[5] M. d'Aquin, A. Adamou, E. Daga, S. Liu, K. Thomas, and E. Motta, "Dealing with diversity in a smart-city datahub," in *Proceedings of the Fifth International Conference on Semantics for Smarter Cities - Volume 1280*, ser. S4SC'14. Aachen, Germany, Germany: CEUR-WS.org, 2014, pp. 68–82. [Online]. Available: http://dl.acm.org/citation.cfm?id=2878779.2878787

[6] S. Pirttikangas, E. Gilman, X. Su, T. Leppnen, A. Keskinarkaus, M. Rautiainen, M. Pyykknen, and J. Riekki, "Experiences with smart city traffic pilot," in *2016 IEEE International Conference on Big Data (Big Data)*, Dec 2016, pp. 1346–1352.

[7] N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2015.

[8] A. I. Maarala, M. Rautiainen, M. Salmi, S. Pirttikangas, and J. Riekki, "Low latency analytics for streaming traffic data with apache spark," in *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, 2015, pp. 2855–2858. [Online]. Available: https://doi.org/10.1109/BigData.2015.7364101

[9] D. PUIU, P. Barnaghi, R. TÖNJES, D. Kumper, M. I. Ali, A. MILEO, J. Parreira, M. Fischer, S. KOLOZALI, N. FARAJIDAVAR, F. Gao, T. IGGENA, T.-L. PHAM, C.-S. NECHIFOR, D. PUSCHMANN, and J. FERNANDES, "Citypulse: Large scale data analytics framework for smart cities," *IEEE Access*, vol. 4, April 2016. [Online]. Available: http://epubs.surrey.ac.uk/810693/

[10] B. Svingen. Publishing with apache kafka at the new york times. Accessed: 2018-01-12. [Online]. Available: https://www.confluent.io/blog/publishing-apache-kafka-new-york-times/

[11] F. Bachmann, L. Bass, P. Clements, D. Garlan, J. Ivers, M. Little, P. Merson, R. Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond*, 2nd ed. Addison-Wesley Professional, 2010.

[12] J. Kohlas and J. Pasquier, "Optimization of spare parts for hierarchically decomposable systems," vol. 8, pp. 294–300, 11 1981.

[13] J. Kreps, L. Corp, N. Narkhede, J. Rao, and L. Corp, "Kafka: a distributed messaging system for log processing. netdb?11," 2011. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.233.1726&rank=1